Adaption offener Systeme durch die Trennung von Daten- und Kontrollfluss

D I S S E R T A T I O N

zur Erlangung des akademischen Grades

doctor rerum naturalium

(Dr. rer. nat.)

im Fach Informatik

eingereicht an der

Mathematisch-Naturwissenschaftlichen Fakultät

Humboldt-Universität zu Berlin

von

Dipl.-Inf. Christian Gierds

Präsident der Humboldt-Universität zu Berlin:
Prof. Dr. Jan-Hendrik Olbertz

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät:
Prof. Dr. Elmar Kulke

Gutachter:
1. Prof. Dr. Wolfgang Reisig
2. Prof. Dr. Karsten Wolf
3. Dr. Matthias Weidlich

Tag der Verteidigung: 12.02.2015

ZUSAMMENFASSUNG

Wir betrachten *offene Systeme*, wobei ein offenes System durch eine *Schnittstelle* und durch *Verhalten* in Form eines Protokolls definiert sei. Interaktion offener Systeme bedeutet für uns den *Austausch von Nachrichten*. Die Interaktion verläuft korrekt, wenn die interagierende offenen Systeme terminieren können.

Die korrekte Interaktion unabhängig entwickelter, offener Systeme ist nicht immer möglich. Obwohl ein offenes Systeme Funktionalität bereitstellt, die ein zweites offenes System benötigt, können *Inkompatibilitäten* in den Schnittstellen oder im Verhalten dazu führen, dass Nachrichten nicht korrekt ausgetauscht werden. Hier kommt die Idee eines *Adapters* zum Tragen, der solche Unterschiede ausgleicht.

Wir führen eine Technik ein, die Inkompatibilitäten der Schnittstelle und des Verhaltens getrennt voneinander betrachtet. Mit Hilfe von *Transformationsregeln* geben wir an, wie Nachrichten zweier offener Systeme im Zusammenhang stehen. Wir modellieren so *korrekten Datenfluss* zwischen den offenen Systemen und überwinden Inkompatibilitäten in der Schnittstelle. Mit Hilfe existierender Techniken zur *Controllersynthese* stellen wir sicher, dass die Transformationsregeln so angewendet werden, dass Inkompatibilitäten im Verhalten überwunden werden. Wir erzeugen *korrekten Kontrollfluss*. Die Einheit aus Transformationsregeln und Controller bezeichnen wir als Adapter.

Aufbauend auf dieser Technik betrachten wir in dieser Arbeit folgende *Fragestellungen*: • Welche Eigenschaften besitzt die vorgestellte Technik? Wann existiert ein Adapter? Wie lässt sich die Technik allgemein auf offene, kommunizierende Systeme anwenden? • Wie verteilen wir einen generierten Adapter auf verschiedene Komponenten? • Welche Informationen können wir bereitstellen, falls wir keinen Adapter finden? • Wie leiten wir ein formales Modell eines Adapters ab, wenn wir nur aufgezeichnetes Verhalten der offenen Systeme, aber keine formalen Modelle gegeben haben?

ABSTRACT

We consider *open systems* being described by an *interface* and *behavior* in form of an protocol. Interaction between open systems means *exchange of messages*. For the interaction to be correct, the interacting open systems must be able to terminate.

The interaction of independently developed open systems is not guaranteed to be correct. Although one open system provides functionality a second open system requires, *incompatiblities* of the interfaces or the behavior may prevent the correct exchange of messages. The idea is to introduce an *adapter* to overcome the incompatiblities.

We present a technique that considers incompatiblities of the interfaces and the behavior separately. With *transformation rules* we specify how messages of two open systems relate to each other. Using these rules we model the *data flow* between open systems and overcome incompatiblities of the interfaces. By using existing techniques for *controller synthesis* we overcome incompatiblities of the behavior. Thus we create correct *control flow*. We call the composition of transformation rules and controller an adapter.

Based on the technique presented in this thesis, we want to answer the following *reseach questions*: • What are the properties of the proposed technique? When does an adapter exist? How can we apply the technique on open, communicating systems in general? • How do we distribute a generated adapter to different components? • Which information can we provide, if adapter synthesis fails? • How can we discover a formal model of an adapter, when we only have recorded behavior of open systems, but no formal models?

INHALTSVERZEICHNIS

1

EINLEITUNG

Was ist ein Adapter? Wann benötigen wir ihn? Warum sollten wir in einem Adapter Datenfluss und Kontrollfluss getrennt betrachten?

1.1 ADAPTER ZUR ÜBERWINDUNG VON INKOMPATIBILITÄTEN ZWISCHEN OFFENEN SYSTEMEN

Wir bezeichnen als *offenes System* ein System, das zum einen über eine *Schnittstelle* und zum anderen über ein *Verhalten* verfügt. Die Schnittstelle dient der Interaktion mit dem System, dem Austausch von Informationen. Die Existenz von Verhalten bedeutet, dass sich das System in mehr als einem Zustand befinden kann.

Diese Definition eines Systems repräsentiert verschiedenste Konzepte. So beschreibt sie bereits fast vollständig einen Service, von dem wir noch fordern, dass er unter einen bestimmten Adresse auffindbar sein muss. Aber auch Objekte in einer Programmiersprache oder technische Geräte erfüllen diese Definition.

Ein Aspekt, der bei offenen Systemen eine wichtige Rolle spielt, ist die Modularisierung und Wiederverwendbarkeit. Ein einzelnes offenes System erfüllt eine gewisse Kernfunktionalität. Für die Erfüllung einer anderen Funktionalität wird ein anderes System benutzt. Um ein komplexes System zu bauen, werden mehrere dieser offenen Systeme *komponiert*. Da bestimmte Funktionalitäten in verschiedensten komplexen Systemen benötigt werden, wird ein einzelnes offenes System immer wieder benutzt, um komplexe Systeme zu bauen.

Ein Gebiet, in dem diese Ideen sehr konsequent gelebt werden, ist das der *Service-orientierten Architekturen* [85]. Ein Service als offenes System wird entwickelt, veröffentlicht und über einen *Servicebroker* an andere Services vermittelt. Durch Komponieren einfacher Services entstehen komplexe Services. Die Services tauschen über *Kanäle* Informationen in Form von *Nachrichten* aus.

Es ist jedoch nicht sicher gestellt, dass wir zwei unabhängig voneinander entwickelte Services einfach komponieren können, auch wenn ein Service die Funktionalität bereit stellt, die der andere benötigt. Es gibt verschiedene Arten von *Inkompatibilitäten*, welche die Komposition verhindern.

Syntax Die Benennung einer Schnittstelle muss für einen anderen Service nachvollziehbar sein. Wenn auf einen bestimmten Teil der Schnittstelle zugegriffen wird, muss klar sein, wie dieser heißt. Unterschiede müssen entsprechend umbenannt werden, um diese Inkompatibilität zu vermeiden.

Semantik Die Bedeutung einer ausgetauschten Nachricht muss für beide Services die gleiche sein. Wenn ein Service eine Liste als Sequenz von Nachrichten sendet, der empfangene Service aber die einzelnen Nachrichten in Form eines Listentyps erwartet, müssen die Nachrichten entsprechend umgewandelt werden. Der Inhalt einer Nachricht muss angepasst werden, wenn die Typen zwar gleich, die Interpretation aber unterschiedlich ist. Eine Nachricht, die eine Längenangabe enthält, muss dann zum Beispiel von Fuß in Meter umgerechnet werden.

Verhalten Ein Service verhält sich unerwartet, wenn er eine Nachricht sendet, die der andere Service gerade nicht erwartet, wenn er eine Nachricht benötigt, die der andere Service gerade nicht senden kann, oder wenn zwei Nachrichten in der falschen Reihenfolge gesendet werden. Um diese Inkompatibilität zu überwinden, ist es möglich, Nachrichten zu erzeugen, zu löschen oder zwischenzuspeichern. Allerdings hängt gerade das Erzeugen und Löschen auch von der Semantik einer Nachricht ab. Wenn ein Service zum Beispiel Anmeldeinformationen eines Nutzers benötigt, können diese in der Regel nicht einfach erzeugt werden.

Nicht-funktionale Eigenschaften Auch wenn Services zusammen korrekt funktionieren, können nicht-funktionale Eigenschaften bewirken, dass wir sie nicht komponieren wollen. Wenn ein Service zum Beispiel eine Antwort in einer gewissen Zeit benötigt, der zweite Service ihm diese Zeit jedoch nicht zusichern kann, dann verzichten wir darauf, die beiden zusammen zu nutzen. Auch der Verbrauch von Ressourcen oder die Zusicherung zum Schutz von Daten können solche Eigenschaften sein.

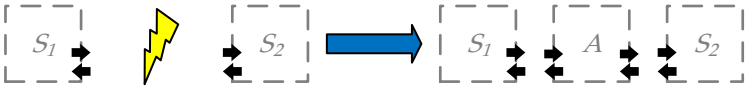


Abbildung 1: Die Service S_1 und S_2 sind inkompatibel (links), können aber mit Hilfe eines Adapters A kommunizieren (rechts).

Wenn zwei Services inkompatibel sind, besteht die Möglichkeit einen oder beide Service entsprechend anzupassen. Dies widerspricht jedoch der Idee eines Service, dass er in seiner Form wiederverwendbar sein soll, also unverändert bleibt.

Die Alternative ist, einen dritten Service, einen *Adapter* [117], einzuführen, der die Inkompatibilitäten überwindet. Der Adapter überwindet die syntaktischen und semantischen Inkompatibilitäten, indem Nachrichten entsprechend umbenannt oder umgewandelt werden. Die Inkompatibilitäten im Verhalten überwindet er durch zwischenspeichern von Nachrichten, solange dies notwendig ist, beziehungsweise durch Erzeugen oder Löschen von Nachrichten.

In Abbildung 1 sehen wir die Veranschaulichung dieser Idee. Die beiden Services S_1 und S_2 können aufgrund von Inkompatibilitäten nicht komponiert werden. Anstelle miteinander zu kommunizieren, lassen wir sie ausschließlich jeweils mit dem Adapter A kommunizieren, der für die entsprechende Kompatibilität sorgt.

Aus der Idee eines Adapters können wir jedoch noch nicht dessen Korrektheit ableiten. Ein Entwickler, der mit der Implementation eines Adapters betraut ist, kann die syntaktischen und semantischen Inkompatibilitäten wahrscheinlich noch überschauen. Wenn das Verhalten der zu adaptierenden Services jedoch etwas komplexer ist, ist es für den Entwickler mühsam, korrektes Verhalten des Adapters zu erreichen.

Mit Hilfe von *Model Checking* [26] kann der Entwickler überprüfen, ob sich ein Adapter korrekt verhält. Ist dies nicht der Fall, bekommt er unter Umständen ein Gegenbeispiel und kann den Adapter anpassen. Dies ist dann allerdings ein iteratives, aufwendiges Verfahren, da normalerweise nicht alle Fehler gleichzeitig entdeckt werden und von Hand ausgebessert werden müssen.

Unser Ziel ist es daher, einen Adapter soweit wie möglich automatisch zu generieren. Da wir verschiedene Arten von Inkompatibilitäten überwinden wollen, definieren wir entsprechende Teile des Adapters, die für jeweils einen Teil der Inkompatibilitäten zu-

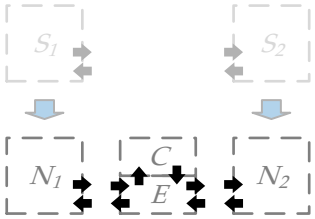


Abbildung 2: Für die formalen Modelle N_1 und N_2 erzeugen wir eine Engine E, welche syntaktische und semantische Kompatibilität sicherstellt, und anschließend automatisch einen Controller C, der korrektes Verhalten sicherstellt.

ständig sind. Auf diese Art können wir bereits bestehende Techniken wiederverwenden und einen Adapter generieren, der vom Entwurf her korrekt ist.

1.2 VORSTELLUNG DER TECHNIK ZUR ADAPTERSYNTHESE

Die Technik in dieser Arbeit konzentriert sich auf zwei Aspekte eines Adapters: semantische Kompatibilität und korrektes Verhalten. Die semantische Kompatibilität schließt auch syntaktische Kompatibilität ein, da dafür einfaches Umbenennen von Nachrichten ausreicht. Dies ist ein sehr spezieller Fall für die Umwandlung verschiedener Nachrichtentypen. Die Inkompatibilitäten nichtfunktionaler Eigenschaften betrachten wir in dieser Arbeit nicht, insbesondere deshalb, weil nicht klar ist, ob ein Adapter Inkompatibilität dieser Art überhaupt überwinden kann.

Unser Ansatz, der in Abbildung 2 gezeigt wird, definiert einen Adapter als zwei Teile: eine *Engine* und einen *Controller*. Die offenen Systeme S_1 und S_2 sind als formale Modelle N_1 und N_2 gegeben.

Die Aufgabe der *Engine* ist die semantische Kompatibilität der Kommunikation sicherzustellen. In der Engine wandelt der Adapter verschiedene Nachrichtentypen in einander um, um so die Nachrichtentypen der offenen Systeme kompatibel abzubilden. Die Engine ist so gestaltet, dass höchstens semantisch kompatibles Verhalten möglich ist. Sie beschreibt somit den *Datenfluss* zwischen den offenen Systemen.

Die Aufgabe des *Controllers* ist das Verhalten der Engine soweit einzuschränken, dass sich die offenen Systeme korrekt verhalten. Er ergänzt somit die Engine um *Kontrollfluss*.

Wir motivieren kurz, welche Möglichkeiten es gibt, Datenfluss und Kontrollfluss automatisch zu erzeugen, und in welcher Form wir dies in dieser Arbeit nutzen. Anschließend gehen wir auf die Fragestellungen ein, die wir mit der vorgestellten Technik in dieser Arbeit beantworten.

1.2.1 *Datenfluss*

Es ist üblich, den Daten- beziehungsweise Nachrichtenfluss zwischen offenen Systemen zu modellieren. Es gibt grundlegende Konzepte, wie eine Nachricht zwischen offenen Systemen geleitet oder manipuliert werden kann. Eine weit verbreitete Sammlung an solchen Konzepten stellen die *Enterprise Integration Patterns* [44] dar. Mit diesen können wir den Datenfluss modellieren und mit einer geeigneten Semantik [31, 93] in ein formales Modell übertragen.

Alternativ stellen Techniken des *Semantic Web* [62] Ansätze bereit, den Datenfluss zwischen zwei offenen Systemen automatisch zu bestimmen. Dies setzt in der Regel voraus, dass ein offenes System semantisch annotiert ist, also die Bedeutung der Daten und Nachrichten zum Beispiel in einer Ontologie erklärt wird. Durch das Vergleichen der Ontologien zweier offener Systeme oder unter Zuhilfenahme weitere Quellen lassen sich Zusammenhänge zwischen den einzelnen Nachrichtentypen ableiten und somit ein Datenfluss zwischen den Systemen beschreiben.

Wir betrachten in dieser Arbeit *Transformationsregeln*. Diese stellen eine Abstraktion der oben genannten Ideen dar. Wir nehmen an, dass wir die Transformationsregeln stets gegeben haben und nicht erst ableiten müssen. Mit den genannten Techniken wäre dies jederzeit möglich, aber nicht im Fokus dieser Arbeit.

1.2.2 *Kontrollfluss*

Allein den Datenfluss zu beschreiben reicht für einen Adapter nicht aus. Die Reihenfolge, in der Nachrichten manipuliert oder weitergeleitet werden, hat einen Einfluss auf das korrekte Verhalten der offenen Systeme.

Den Kontrollfluss erzeugen wir automatisch. In der Controllersynthese [61, 87] wird alles inkorrekte Verhalten ausgeschlossen.

Wir synthetisieren einen Controller als eigenen Teil des Adapters, der direkt den Datenfluss in der Engine steuert und somit indirekt das korrekte Verhalten der offenen Systeme sicherstellt.

1.3 FRAGESTELLUNGEN

Im Rahmen dieser Arbeit beantworten wir folgende Fragen, wobei die Trennung von Datenfluss und Kontrollfluss zentraler Aspekt ist.

1. Wie setzen wir formal die Adaptersynthese um, sodass Daten- und Kontrollfluss von einander getrennt sind? (Kapitel 3)
2. Welche Eigenschaften hat die Technik? (Kapitel 4)
3. Wie setzen wir die Technik um, um existierende offene Systeme zu adaptieren? (Kapitel 5)
4. Wie können wir einen synthetisierten Adapter auf mehrere Komponenten verteilen? (Kapitel 6)
5. Welche Informationen können wir bereitstellen, falls die Adaptersynthese fehlschlägt? (Kapitel 7)
6. Können wir ein formales Modell eines Adapters erstellen, auch wenn wir für die offenen Systeme keine formalen Modelle kennen? (Kapitel 8)

1.4 STAND DER FORSCHUNG

Der Begriff *Adapter* hat seine Ursprünge im komponentenbasierten Softwareentwurf [116, 117] beziehungsweise in der objektorientierten Programmierung [34]. Hier steht bereits im Vordergrund, eine existierende Komponente wiederverwenden zu können. Unter Umständen müssen dazu aber die Schnittstelle, also der Zugriff auf Datentypen und Funktionen, und das Verhalten, wenn die Komponente verschiedene Zustände unterscheidet, angepasst werden. In Zusammenhang mit Service-orientierten Architekturen [85] wurde das Thema in zahlreichen Arbeiten vertieft [97]. Aber auch im Rahmen der Prozessintegration [66] ist das Thema aufgegriffen worden.

Die Arbeiten lassen sich in drei Kategorien einordnen: Kompatibilität bezüglich der Schnittstelle herstellen, Kompatibilität bezüglich des Verhaltens herstellen oder beides.

Nur Schnittstelle adaptieren

Inkompatibilitäten in der Schnittstelle betreffen syntaktische und semantische Unterschiede. Um die Schnittstellen kompatibel zu bekommen, beschreiben wir, wie Typen in der ersten Schnittstelle mit Typen in der zweiten Schnittstelle zusammenhängen. Diese Zuordnung kann manuell stattfinden, allerdings gibt es Ansätze, die Zuordnung automatisch zu finden.

Eine mögliche Idee besteht im Vergleich komplexer Datentypen. Da die Basisdatentypen in einer Sprache fix sind, wird jeder komplexe Datentyp als Struktur über den Basisdatentypen beziehungsweise transitiv über andere komplexe Datentypen beschrieben. Sind zwei komplexe Datentypen gleich aufgebaut, beschreiben sie wahrscheinlich die gleich Art Daten.

Auf dieser Grundlage basiert das *Schema Matching* [89], das Schemata der *Extensible Markup Language* (XML) [107] vergleicht und eine Zuordnung ausrechnet. Gerade im Bereich von Webservices ist XML als Basis zur Beschreibung von Schnittstellen weit verbreitet, insbesondere in Form der *Web Service Description Language* (WSDL) [25].

Ponnekanti und Fox [88] schlagen ein Framework vor, um strukturelle und semantische Inkompatibilitäten, insbesondere bezüglich Werten und deren Kodierung, zu überwinden.

Fuchs [33] definiert *Service Views*. Für die WSDL-Beschreibung eines Service erzeugt er verschiedene Sichten, sodass der Service unterschiedlichen Partnern zur Verfügung steht. Ähnlich betrachten Cavallaro u. a. [24] ausschließlich die WSDL-Beschreibungen von Services, um auf ihnen Adapter zu definieren.

Neben dem strukturellen Matching von Schnittstellen gibt es viele Ansätze zur Adaptersynthese, die auf Technologien des *Semantic Web* [11] zurückgreifen. Anstelle nur die Struktur zu betrachten, wird die semantische Bedeutung der Datentypen betrachtet.

Ein offenes System muss dazu um eine semantische Beschreibung ergänzt werden. Durchgesetzt hat sich die semantisch Beschreibung in Form von Ontologien mittels der *Semantic Ontology Web Language* (OWL-S) [62].

Neben den Ansätzen, welche die semantische Analyse der Schnittstellen integrieren, um Adapter mit Verhalten zu generieren, beschreibt zum Beispiel Elgedawy [30] eine rein semantische Analyse und Zuordnung zwischen Schnittstellen. Gerade für zustandslose Systeme ist solch eine Analyse ausreichend, um sie zu adaptieren.

Alternativ betrachten Kongdenfha u. a. [50] offene Systeme mit *Aspekten*, d. h., solche Systeme sind entsprechend annotiert und werden mit aspektorientierten Techniken adaptiert.

In der in dieser Arbeit beschriebenen Technik nehmen wir eine semantische Spezifikation in Form von Transformationsregeln als gegeben an. Neben der Möglichkeit, die Spezifikation von Hand anzugeben, bieten die oben beschriebenen Techniken die Möglichkeit, eine semantische Spezifikation automatisch abzuleiten, indem wir mit diesen Techniken semantische Kompatibilität für die Schnittstellen zweier zu adaptierender Systeme herstellen.

Nur Verhalten adaptieren

Zwei offene Systeme sind verhaltenskompatibel, wenn die richtigen Nachrichten zur richtigen Zeit gesendet und empfangen werden. Sie sind somit inkompatibel, wenn: ein System eine Nachricht sendet, die das andere nicht erwartet; ein System ein Nachricht nicht sendet, die das andere erwartet; oder Nachrichten in der falschen Reihenfolge gesendet werden.

Die beschriebenen Probleme treten in der Regel deshalb auf, weil die zugrunde liegenden Formalismen wie Prozessalgebra oder Automaten mit einer synchronen Kommunikationssemantik arbeiten. Daher müssen Nachrichten genau in der Reihenfolge empfangen werden, in der sie gesendet werden.

In ersten Arbeiten, wie zum Beispiel von Benatallah u. a. [10], werden Muster definiert, wann zwei Systeme inkompatibles Verhalten zeigen und wie dies korrigiert werden kann. Der Vorteil dieser Methode ist die einfache Anwendbarkeit bei geringer Komplexität. So setzen La und Kim [53] und Kongdenfha u. a. [51] diese Arbeit fort, indem sie Inkompatibilitätsmuster beziehungsweise Adaptionsmuster definieren.

Für den Fall, dass der Zeitpunkt des Sendens und Empfangs einer Nachricht beliebig weit auseinander liegen kann, und einfaches Vertauschen zweier Nachrichten nicht ausreicht, übernimmt der Adapter die Aufgabe, eine Nachricht solange zu puffern, bis diese benötigt wird.

Kumar und Shan [52] betrachten anfänglich einen kleinen Satz an Entwurfsmustern, aus denen ein Service aufgebaut sein darf. Sie identifizieren, wann zwei Services mit Hilfe eines Pufferprozesses adaptiert werden können, und leiten solch einen Prozess ab.

In späteren Arbeiten liegt der Fokus auf *optimalen* Adaptern, d. h., auf Adaptern, die möglichst wenig Nachrichten puffern müssen. Seguel, Eshuis und Grefen [98] betrachten minimale Adapter für den Fall, dass die offenen Systeme asynchron über eine Warteschlange kommunizieren, und die Reihenfolge in der Warteschlange verändert werden muss. Shan und Kumar [99] erweitern ihren früheren Ansatz, indem sie mit Hilfe linearer Optimierung einen optimalen Pufferprozess bestimmen.

Einen genetischen Ansatz verfolgen Belle, Mens und D'Hondt [9]. Sie sind somit nicht auf bestimmte Entwurfsmuster beschränkt und können allgemeineres Verhalten adaptieren.

In dieser Arbeit nehmen wir ein asynchrones Kommunikationsmodell an. In diesem können sich Nachrichten beliebig überholen und werden solange gepuffert, bis diese empfangen werden. Die beschriebenen Probleme ergeben sich in unserem Kommunikationsmodell somit nicht.

Die genannten Arbeiten betrachten allerdings ausschließlich Systeme, die nicht von selbst in einen schlechten Zustand gelangen können. Solange die von einem System gesendeten Nachrichten empfangen und eine der vom System zu empfangen Nachrichten gesendet wird, entsteht kein schlechtes Verhalten. Wir betrachten jedoch Systeme, in denen eine Nachricht auch zur falschen Zeit empfangen werden kann, d. h., ein System empfängt diese Nachricht zwar, aber das führt intern zu einem schlechten Zustand. Daher reicht es in unserem Fall nicht aus, Nachrichten solange zu puffern, bis sie empfangen werden können, sondern der Adapter muss sicherstellen, dass der Empfang einer Nachricht durch ein System nicht zu einem schlechten Zustand führt. Wir untersuchen daher in unserer Technik den Raum der erreichbaren Zustände der Systeme und schließen Verhalten im Adapter aus, das zu einem schlechten Zustand führt.

Wir sind dabei nicht auf bestimmte Muster beschränkt. Die von uns betrachteten Systeme dürfen beliebig strukturiert sein, und wir können beliebiges inkompatibles Verhalten adaptieren, wobei nur von der Semantik der Nachrichten abhängt, ob dies möglich ist.

Schnittstelle und Verhalten adaptieren

Es gibt viele Ansätze, die mit Hilfe eines Adapters sowohl semantische als auch Verhaltenskompatibilität zweier offener Systeme herstellen. Die ersten Ideen beschränken sich noch darauf, bestimmte Muster an inkompatiblen Verhalten zu adaptieren, während spätere Ansätze den Zustandsraum betrachten, und entweder schlechtes Verhalten entfernen oder durch weitere Regeln auflösen.

Benatallah u. a. [10] benutzen Schema Matching um semantische Kompatibilität herzustellen und beschreiben dann bestimmte Muster, für die sie Verhaltenskompatibilität in Transitionssystemen herstellen können.

Andere Autoren benutzen Prozessalgebra, um ihre Systeme zu modellieren. Dies hat den Vorteil, dass sich die Zuordnungen der Schnittstellen als prozessalgebraische Ausdrücke schreiben und einfach komponieren lassen. Bracciali, Brogi und Canal [13] nutzen den π-Kalkül [70] für die Modellierung von Softwarekomponenten und bauen für diese iterativ korrektes Verhalten auf, das die Schnittstellenzuordnung bewahrt. Um *Web Service Choreography Interface* (WSCI) [109], eine Sprache zur Orchestrierung von Webservices, zu formalisieren, benutzen Brogi u. a. [16] den *Calculus of Communicating Systems* (CCS) [69]. Sie greifen dann aber auf die Resultate von Bracciali, Brogi und Canal [13] zurück. Einen ähnlichen Ansatz verfolgen Brogi, Canal und Pimentel [14] dann für Softwarekomponenten. Deren Schnittstelle erweitern sie um Protokolle, die das Verhalten einer Komponente beschreiben, und um eine High-Level-Notation für die Datentypen.

Eine graphische Notation für typische Transformationsregeln geben Dumas, Spork und Wang [28] an. Ihr Ziel ist es jedoch nicht, zwei Systeme zu adaptieren, sondern für ein System anstelle der tatsächlichen gegebenen Schnittstelle eine andere benötigte Schnittstelle nach außen darzustellen. Mit Hilfe der von ihnen eingeführt Notation geben sie an, in welchen Zustandsübergang welche Regel angewendet wird.

Motahari Nezhad u. a. [78] benutzen zuerst Schema Matching, das sie mit Information aus den WSDL-Dateien der Systeme anreichern. Sie bauen dann einen Verhaltensbaum auf, in dem sie nach Verklemmungen suchen, in denen keine brauchbare Transformation zur Verfügung steht. Die Transformation muss dann von einem Nutzer bereitgestellt werden. Der Ansatz untersucht somit den Zustandsraum auf schlechte Zustände. Er vermeidet diese jedoch nicht, sondern sie müssen manuell aufgelöst werden.

Nach einer Analyse der Schnittstelle untersuchen Motahari Nezhad, Xu und Benatallah [76] das Verhalten der gegebenen Systeme, um Anforderungen bezüglich der Reihenfolge von Nachrichten zu verbessern.

Es existieren noch weitere Ansätze, die korrektes Verhalten mit alternativen Methoden sicher stellen. Martín und Pimentel [63] definieren Heuristiken und benutzen Expertensysteme, um eine Adapterspezifikation automatisiert zu finden. Beauche und Poizat [8] beschreiben die Adaptersynthese als Planungsproblem, wobei der Adapter nur Daten transformiert und das Planungsproblem ein Auswahlproblem löst, welche Services in der Gesamtkomposition enthalten sein sollen. Canal, Poizat und Salaün [21] entwickeln modellbasiert und Mateescu, Poizat und Salaün [65] constraintbasiert Adapter. Allerdings muss in beiden Fällen anschließend verifiziert werden, ob sich die Adapter korrekt verhalten. Cao und Nymeyer [22] müssen ihren Adapter auch verifizieren, allerdings verbinden sie dies mit einer Markovanalyse, um einen optimalen Adapter zu generieren.

Im Gegensatz zu den zuvor genannten Ansätzen, die einen Adapter zur Entwurfszeit betrachten, gibt es auch Ansätze, wie offene Systeme zur Laufzeit adaptiert werden können. Zhou u. a. [119] klinken sich in die Laufzeitumgebung ein, um die Schnittstellenzuordnung umzusetzen. Sie erlauben dabei jedoch nur 1-zu-1-Zuordnungen zwischen den Schnittstellentypen. In der „Adaptation Machine“ von Wang u. a. [110] sind allgemeinere Transformationsregeln zulässig. Sie betrachten den aktuellen Zustand der offenen Systeme, und wählen eine der vorgegebenen Transformationsregeln aus, die im aktuellen Zustand die Schnittstellen sinnvoll adaptiert.

Eine Gebiet, das mit dem der Adaptersynthese verwandt ist, ist das der *Service Orchestrierung* [86]. Ziel der Service Orchestrierung ist, einen zentralen Service, den Orchestrator, zu erzeugen, sodass die orchestrierten Services eine vorgegebene Spezifikation erfüllen. Ein Adapter ist von seiner Aufgabe her mit einem Orchestrator verwandt, da er zentral die gegebenen Services steuert. Allerdings ist der Fokus in der Service Orchestrierung ein anderer als in der Adaptersynthese. In Service Orchestrierung stellt der Orchestrator sicher, dass die Komposition der Services eine vorgegebene Spezifikation eines Gesamtsystems erfüllt, indem er mit den einzelnen Services zu den entsprechenden Zeitpunkten interagiert. Ein Adapter konzentriert sich auf den korrekten Austausch von Nachrichten, ohne dabei auf eine spezielle Struktur des Gesamtsystems zu schauen.

Die oben aufgezählten Ansätze haben gemeinsam, dass sie den Datenfluss und den Kontrollfluss eines Adapters als zwei getrennte Problemdimensionen wahrnehmen. In der Umsetzung trennen sie diese beiden Aspekte allerdings nicht mehr. Zwar wird teilweise der Zustandsraum betrachten und schlechtes Verhalten entfernt, beziehungsweise werden die Transformationen so zu einem Adapter zusammengesetzt, dass kein inkorrektes Verhalten auftreten kann. In beiden Fällen ist der Adapter jedoch monolithisch, sodass ein Ausnutzen der einzelnen Dimensionen nicht oder nur schwer möglich ist.

In dieser Arbeit präsentieren wir einen Ansatz, der Daten- und Kontrollfluss nicht nur konzeptionell getrennt betrachtet, sondern die beiden Aspekte getrennt im Adapter umsetzt. Dies ermöglicht es uns, weiterführende Fragestellungen, die nur den Datenfluss oder nur den Kontrollfluss betreffen, zu betrachten.

Diese Flexibilität nutzen Mooij und Voorhoeve [73] aus und beschreiben den Datenfluss mit Datenbankoperationen.

1.4.1 *Praktische Umsetzung*

Wir setzen die in dieser Arbeit vorgestellten Techniken in verschiedenen Werkzeugen um. Aber auch andere Arbeiten resultieren in entsprechenden Werkzeugen, welche wenigstens die Machbarkeit der vorgestellten Ansätze zeigt.

Motahari Nezhad u. a. [78] implementieren die Idee des Schema Matchings und der Analyse des Verhaltensbaums im *IBM WebSphere Integration Developer*. Cámara u. a. [20] stellen eine Toolbox mit dem Namen *Itaca* bereit. Neben der Möglichkeit, Adapter zu spezifizieren, erlaubt die Toolbox auch Simulation, Verifikation und die Generierung von Adaptern auf Basis einer Spezifikationen. Brogi und Popescu [15] überführen BPEL-Prozesse nach YAWL, einem Werkzeug zur Ausführung von Workflows. In YAWL wird ein Adapter generiert und auf Verklemmungen überprüft. Ist keine Verklemmung erreichbar, wir der Adapter nach BPEL übersetzt.

Für die algorithmischen Ergebnisse dieser Arbeit ist jeweils ein eigenes Werkzeug entstanden. So sind wir in der Lage, einen Adapter zu synthetisieren, auszuführen, zu verteilen, bei Problemen mit der Synthese zu diagnostizieren, die Existenz eines adaptierbaren Systems zu überprüfen, und ein formales Adaptermodell abzuleiten, wenn uns für die offenen Systeme nur aufgezeichnetes Verhalten zur Verfügung steht.

1.4.2 *Verteilen*

Wir betrachten in dieser Arbeit die Möglichkeit, einen synthetisierten Adapter auf verschiedene Komponenten zu verteilen. Existierende Arbeiten betrachten diese Aufgabe insbesondere im Zusammenhang der Orchestrierung von Services.

Salaün [96] betrachtet die Verteilung eines Orchestrator in lokale Wrapper. Jeder Service bekommt einen lokalen Orchestrator, der den Service steuert, und dafür Sorge trägt, dass benötigte Nachrichten mit anderen Wrappern ausgetauscht werden. Der Orchestrator tauscht lediglich Nachrichten aus, ohne sie semantisch zu transformieren. Zudem hängt die Verteilung von den vorgegebenen Services ab und kann nicht weiter beeinflusst werden. Autili u. a. [6] verfolgen einen ähnlichen Ansatz. Auch hier wird ein monolithischer Adapter auf die Services aufgeteilt. Dafür ist die Einführung zusätzlicher Nachrichten notwendig, um die einzelnen Komponenten zu synchronisieren. Melliti, Poizat und Mokhtar [67] generieren zunächst einen zentralen Adapter, den sie anschließend verteilen. Auch sie verteilen den Adapter als Wrapper für die beteiligten Services, die sich über zusätzliche Nachrichten synchronisieren müssen.

Das Ziel der existierenden Arbeiten ist somit, einen Adapter so zu verteilen, dass jeder Service den Teil ausführt, der seinen Teil des Nachrichtenaustauschs orchestriert. Die Verteilung verursacht zusätzliche Synchronisation zwischen den Komponenten. Wir dagegen betrachten eine Verteilung des Adapters auf beliebige Komponenten, wobei wir eine Verteilung der Transformationsregeln vorgeben können. Unsere Verteilung ist maximal und benötigt keine zusätzliche Synchronisation zwischen den Komponenten. Allerdings können wir einen Adapter nicht immer verteilen, weil dies durch die genutzten Techniken auf Petrinetzbasis nicht garantiert werden kann.

Wir bauen auf der Arbeit von Glabbeek, Goltz und Schicke [40] auf, die Petrinetz asynchron verteilen, indem sie Transitionen und Plätze einführen, die wir zur Kommunikation zwischen den Komponenten nutzen können. Dieser Ansatz ist etwas genereller als die Ideen von Mennicke, Oanea und Wolf [68] und Lehmann [54], die davon abhängen, dass entsprechende Strukturen bereits in einem Netz vorhanden sind.

Da wir beliebige Strukturen zulassen, lassen sich zum Beispiel Resultate für Free-Choice-Petrinetze [12] und verwandte Netzarten [1] nicht übertragen. Wir bauen jedoch auf einem recht allgemeinen Formalismus zum Verteilen von Petrinetzen auf, dessen Äquivalenzbetrachtungen es uns erlauben, flexibel bei der Wahl des Korrektheitskriterium zu bleiben, und somit viele Adapter zu verteilen.

1.4.3 *Diagnose*

Ziel der Diagnose ist zu verstehen, warum wir keinen Adapter synthetisieren können. Wie bei der Adaptersynthese selbst betrachten erste Ansätze von Li u. a. [55] und Taher u. a. [105] verschiedene Muster, die sie in einer Eingabe identifizieren und als Ursache aufzeigen. Auch der Ansatz von Motahari Nezhad u. a. [78] beruht am Ende auf Diagnose. Im Verhaltensbaum werden die Stellen lokalisiert, in denen eine Verklemmung auftritt, die durch zusätzliche Transformationsregeln aufgelöst werden muss. Vaculín und Sycara [106] betrachten für Systeme, die in OWL-S spezifiziert sind, Abläufe und wie sich auf diesen Abläufen Transformationsregeln anwenden lassen. Sie betrachten jedoch nicht den Fall, warum die Adaptersynthese nicht gelingt.

Keiner der existierenden Ansätze betrachtet explizit das mögliche Interaktionsverhalten mit den gegebenen offenen Systemen. Wir vergleichen es hingegen mit dem beobachtbaren Verhalten während der Adaptersynthese und können so jeden Zustand bestimmen, in dem zusätzlichen Verhalten möglich ist. Die existierenden Ansätze beschränken sich auf Verklemmungen, während wir Korrektheitskriterien betrachten können, in dem sich das korrekte Kommunikationsverhalten eines offenen Systems beschreiben lässt.

1.4.4 *Discovery*

In der *Process Discovery* [2] ist das Ziel, ein formales Modell eines Prozesses zu bestimmen, das in einem Log aufgezeichnetes Verhalten möglichst gut erklärt. Die *Service Discovery* [3] erweitert diese Idee explizit auf kommunizierende Systeme.

In dieser Arbeit ist das Ziel, aus Patterns, die die Bausteine eines Adapters bilden, ein Modell eines Adapters zu konstruieren, welches das aufgezeichnete Verhalten von Services, die adaptiert werden, gut erklärt. Es existiert also eine Implementation eines Adapters, aber keine formales Modell. Dieses leiten wir ab, um so Aussagen über den Adapter treffen zu können.

In Arbeiten [41, 42, 79, 81, 82, 95] zur Service Discovery werden Modelle abgeleitet, die sich ausschließlich auf das aufgezeichnet Verhalten beziehen und somit keine Services in Betracht ziehen, die umfangreiches internes oder nicht-beobachtbares Verhalten zeigen.

Die Idee, erst Modelle der Services abzuleiten und auf diese dann die Adaptersynthese anzuwenden, verfolgen wir nicht. Wir können zwar die Güte der einzelnen Servicemodelle bestimmen, jedoch haben wir so keinen Einfluss auf die Güte des Adaptermodells. Zudem ist es möglich, dass das Verhalten der Servicemodelle so inkompatibel ist, dass wir nicht in der Lage sind, einen Adapter zu synthetisieren. Dies widerspricht aber der Voraussetzung, dass der Adapter bereits implementiert ist.

Die Idee, Patterns so zu kombinieren, dass sie eine Anforderung erfüllen, ist eng verwandt mit dem Thema der *Servicekomposition* [90]. In der Servicekomposition werden existierende Service so komponiert, dass sie eine vorgegebene Spezifikation erfüllen. Allerdings liegt der Fokus hier auf zustandslosen Services [118], die zu einer bestimmten Struktur zusammengefügt werden. Für die einzelnen Services ist es nicht möglich, dass das Verhalten inkorrekt ist, was wir jedoch in Betracht ziehen müssen. Zudem können wir keine formale Spezifikation des Adapters angeben, diese wollen wir ja erst ableiten.

Da die Patterns zustandsbehaftet sind, bietet die *Serviceorchestrierung* [19] Impulse. Wie oben beschrieben, ist es Ziel der Orchestrierung eine gegebene Menge von Services gemäß einer Spezifikation über einen zu synthetisierenden Orchestrator zu steuern. Jedoch werden in unserem Fall nicht die Pattern orchestriert, sondern sie bilden den Orchestrator.

1.5 Wissenschaftlicher Beitrag

Bezüglich der Fragestellungen in Abschnitt 1.3 leistet diese Arbeit folgenden Beitrag:

1. Wir beschreiben eine Technik zur Adaptersynthese, die Datenfluss und Kontrollfluss nicht nur konzeptionell unterscheidet, sondern auch in einem Adapter getrennt umsetzt. Diese Trennung im Adapter geht über die konzeptionell Trennung bestehender Arbeiten hinaus. (Kapitel 3)

2. Wir zeigen, unter welchen Bedingungen ein Adapter existiert, und dass die Existenz ausschließlich semantischen Einschränkungen unterliegt. (Kapitel 4)

3. Analog zu existierenden Ansätzen zeigen wir, wie wir Geschäftsprozesse, die in BPEL beschrieben sind, adaptieren, und wie wir den erzeugten Adapter mit den Prozessen interagieren lassen. Dazu stellen wir das Werkzeug Marlene vor,

das die in Kapitel 3 Technik umsetzt, und das Werkzeug CHARLOTTE, welches einen Adapter getrennt nach Datenfluss und Kontrollfluss ausführt und mit den Geschäftsprozessen interagieren lässt. (Kapitel 5)

4. Aufbauend auf der Arbeit von Glabbeek, Goltz und Schicke [40] zum Verteilen von Petrinetzen entwerfen wir einen Algorithmus zum Verteilen von Adaptern. Dazu beschreiben wir, wie die initiale Verteilungsvorschrift für den Datenfluss auf den gesamten Adapter erweitert werden muss. (Kapitel 6)

5. Für den Fall, dass wir keinen Adapter mit der vorgestellten Technik finden, beschreiben wir einen Ansatz, um zusätzliches Interaktionsverhalten in einem Adapter zu ermöglichen. Wir benutzen dazu die Beschreibung des Interaktionsverhaltens [61] der gegebenen offenen Systeme, um genau die Stellen zu identifizieren, an denen ein Adapter mehr Verhalten zeigen kann. (Kapitel 7)

6. Für aufgezeichnetes Verhalten offener Systeme und einer vorgegebenen Menge an Bausteinen definieren wir Maße, wie gut ein aus den Bausteinen aufgebauter Adapter das aufgezeichnet Verhalten erklären kann. Das Werkzeug PETRA bestimmt ein möglichst gutes Modell eines Adapters bezüglich der definierten Maße. (Kapitel 8)

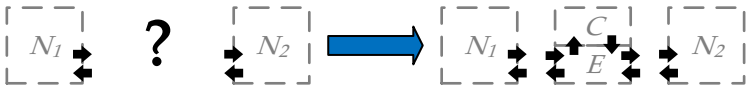
1.6 GLIEDERUNG

Nachdem wir grundlegende Formalismen in Kapitel 2 eingeführt haben, untergliedert sich die Arbeit in zwei Teile. In Teil I, welcher die Kapitel 3 bis 5 umfasst, betrachten wir eine Technik zur Adaptersynthese, deren Eigenschaften und deren Implementation. In Teil II, welcher die Kapitel 6 bis 8 umfasst, zeigen wir, wie wir auf Basis der vorgestellten Technik einen Adapter verteilen, Diagnoseinformationen ermitteln, falls wir keinen Adapter synthetisieren können und ein formales Modell eines Adapters ableiten, wenn uns keine formalen Modelle der offenen Systeme zur Verfügung stehen. Wir fassen die Ergebnisse in Kapitel 9 zusammen und zeigen mögliche weitere Fragestellungen auf.

FORMALE GRUNDLAGEN In Kapitel 2 führen wir die Formalismen ein, die für das weitere Verständnis der arbeitet notwendig sind. Wir definieren *offene Netze* als Modellformalismus für offene Systeme. Wir geben einen konkreten Ansatz zur

Controllersynthese an, der zu einem gegebenen offenen Netz den *allgemeinsten Kommunikationspartner* berechnet.

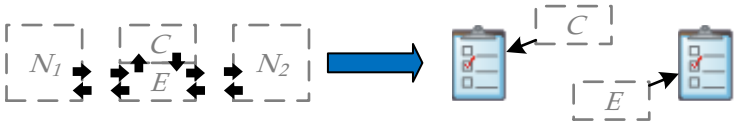
ADAPTERSYNTHESE Kapitel 3 beinhaltet die zentrale Technik zur Adaptersynthese. Wir formalisieren den Begriff der *Transformationsregel* zur Beschreibung des Datenflusses.



Für N_1 und N_2 als formale Modelle offener Systeme definieren eine *Engine* E, welche den Datenfluss zwischen N_1 und N_2 auf Basis von Transformationsregeln beschreibt. Wir benutzen den in Kapitel 2 beschriebenen Ansatz zur Controllersynthese, um die Engine mit einem Controller C zu einem vollwertigen Adapter zu ergänzen.

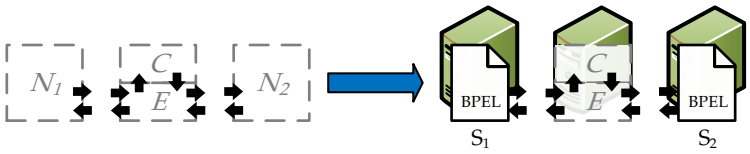
In den folgenden Kapiteln nutzen wir die Kenntnis über den Aufbau des Adapters aus Engine und Controller gezielt aus.

EIGENSCHAFTEN DER TECHNIK Die Eigenschaften der vorgestellten Techniken betrachten wir in Kapitel 4.



Dabei interessiert uns, unter welchen Umständen ein Adapter existiert, und wie wir das Konzept der Engine E und des Controllers C ausnutzen können, um die Adaptersynthese zu beeinflussen.

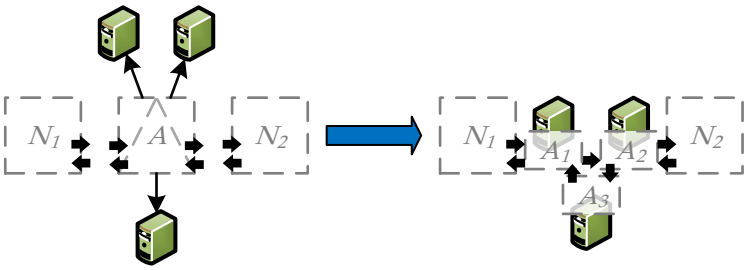
PRAKTISCHE UMSETZUNG In Kapitel 5 beschreiben wir die Implementation der Technik zu Adaptersynthese.



Zum einem implementieren wir genau den Ansatz, wie er in Kapitel 3 beschrieben ist, d. h., für die formalen Modelle N_1 und N_2 stellen wir ein Werkzeug bereit, welches die Engine E und den Controller C synthetisiert.

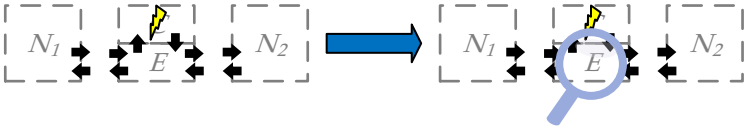
Zum anderen stellen wir eine Implementation vor, welche die Ausführung von E und C gestattet. So sind wir in der Lage, existierende Geschäftsprozesse S_1 und S_2 zu adaptieren.

VERTEILEN In Kapitel 6 beschreiben wir, wie wir einen synthetisierten Adapter maximal auf Komponenten verteilen.



Ausgangspunkt ist eine vorgegebene Verteilung der Transformationsregeln und somit des Datenflusses auf Komponenten. Ausgehend von dieser Verteilung bestimmen wir eine Verteilung des Kontrollflusses, wenn dies möglich ist. Falls eine Verteilung möglich ist, geben wir die einzelnen Komponenten als offene Netze an.

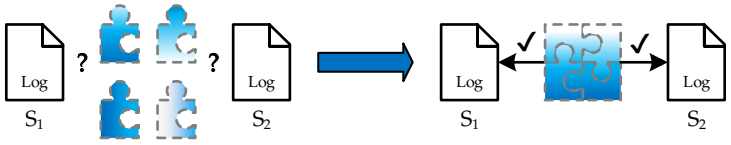
DIAGNOSE In Kapitel 7 betrachten wir den Fall, dass wir mit der Technik aus Kapitel 3 keine Adapter synthetisieren können, weil die Controllersynthese aus Kapitel 2 fehlschlägt.



Wir nutzen in diesem Fall aus, dass wir die Struktur der Engine kennen, und die Engine als einzigen Ort eines Fehlers festlegen können. So definieren wir Situationen, in denen eine Engine mehr Verhalten hätte zeigen können in der

Interaktion mit N_1 und N_2. So erhalten wir Hinweise, wie eine Transformationsregel aussehen muss, die wir zusätzlich betrachten müssen, um einen Adapter zu synthetisieren.

Discovery In Kapitel 8 betrachten wir eine Möglichkeit, ein formales Modell eines Adapters abzuleiten, wenn wir für die zu adaptierenden System S_1 und S_2 keine formalen Modelle gegeben haben, und somit die Technik aus Kapitel 3 nicht anwenden können.



Wir nehmen an, dass S_1 und S_2 ihr Verhalten jeweils in einem *Log* aufzeichnen, wenn sie ausgeführt werden. Für den Adapter nehmen wir an, dass wir grundlegende Bausteine in Form von Patterns kennen, aus denen er aufgebaut ist. Wir beschreiben, wie wir mögliche Adaptermodelle aus den Patterns ableiten und bewerten die *Güte* eines Modells über die Möglichkeit, das Verhalten in den Logs der offenen Systeme zu erklären.

2

FORMALE GRUNDLAGEN

Wie beschreiben wir offene Systeme und deren Interaktion?

2.1 PROBLEMSTELLUNG

In diesem Kapitel führen wir existierende Notationen und Formalismen ein, die für diese Arbeit die Grundlage bilden. Dazu werden wir insbesondere *offene Netze* als Modellformalismus beschreiben und anschließend auf die *Controllersynthese* für offene Netze eingehen. Die hier eingeführten Definitionen werden wir in allen folgenden Kapiteln durchgehend nutzen.

Offene Netze sind eine Erweiterung von Petrinetzen und erlauben, das Verhalten von offenen Systemen zu beschreiben. Alternativ wären auch endliche Automaten [57] oder andere Formalismen wie Prozessalgebra [69] möglich, um das Verhalten zu modellieren. Allerdings liegt in dieser Arbeit ein starker Fokus auf der Komposition von offenen Systemen, und die Transformationsregeln im nächsten Kapitel lassen sich einfach als Petrinetztransition verstehen, sodass wir Petrinetze benutzen, um Systeme zu modellieren.

Die Controllersynthese berechnet einen *allgemeinsten Kommunikationspartner* zu einem offenen Netz N bezüglich eines vorgegebenen Korrektheitskriterium. Um den allgemeinsten Kommunikationspartner zu erhalten, überapproximieren wir das korrekte Kommunikationsverhalten mit N und entfernen in der Überapproximation iterativ inkorrektes Verhalten bis der allgemeinste Kommunikationspartner übrig bleibt, sofern er existiert.

GLIEDERUNG Zuerst wollen wir einige grundlegende Definitionen zu Mengen und Funktionen in Abschnitt 2.2 festlegen. Anschließend führen wir in Abschnitt 2.3 offene Netze und deren Komposition ein. In Abschnitt 2.4 betrachten wir eine Technik zur

Controllersynthese, die einen allgemeinsten Kommunikationspartner, d. h.einen Partner mit größtmöglichem Verhalten, erzeugt.

2.2 GRUNDLEGENDE DEFINITIONEN

Zunächst benötigen wir die Menge der *natürlichen Zahlen* $\mathbb{N}$ einschließlich 0: $\mathbb{N} = \mathbb{N}_0 = \{0, 1, 2, \ldots\}$.

Eine *Multimenge* M über einer Trägermenge X definiert eine Funktion $M : X \to \mathbb{N}$, die jedem Element in X dessen Häufigkeit zuordnet. Jede Menge $Y \subseteq X$ impliziert eine Multimenge mit $Y(y) = 1$, falls $y \in Y$, und $Y(y) = 0$ sonst.

Ähnlich wie für Mengen, können wir auch für Multimengen eine *Teilmengenbeziehung* definieren. Für zwei Multimengen M und N über der gemeinsamen Trägermenge X gilt $M \subseteq N$ genau dann, wenn $M(x) \leq N(x)$ für alle $x \in X$.

Wir werden folgende Schreibweisen im weiteren Verlauf benutzen:

- $\{a, b, c\}$ beschreibt die *Menge* der Elemente a, b, c,
- $\mathcal{P}(X)$ beschreibt die Potenzmenge über X, also die Menge aller Teilmengen von X,
- $[a, a, b, b, b]$ beschreibt die *Multimenge* durch Auflistung der Element gemäß ihrer Anzahl mit $M(a) = 2$, $M(b) = 3$ und $M(x) = 0$ für alle anderen $x \in X$,
- $X \times Y$ beschreibt das *Kreuzprodukt* zweier Mengen X und Y; dessen Elemente $\langle x, y \rangle \in X \times Y$ sind *Tupel*,
- $f_{|Y} : Y \to Z$ beschreibt die *Beschränkung einer Funktion* $f : X \to Z$ auf eine Teilmenge $Y \subseteq X$, mit $f_{|Y}(y) = f(y)$ für $y \in Y$, und
- $\vec{a}$ beschreibt eine möglicherweise unendliche Sequenz, wobei wir die einzelnen Elemente als $a_1, a2, \ldots$ aufzählen.

2.3 OFFENE NETZE

Offene Netze sind eine Erweiterung von Petrinetzen und unser Mittel der Wahl, um offene Systeme zu modellieren. Unser Fokus liegt auf endlichen Systemen. Somit bieten

sich verschiedene Formalismen wie zum Beispiel endliche Automaten zum Modellieren von Verhalten an. Allerdings bieten Petrinetze gerade im Zusammenhang mit der Komposition offener Systeme Vorteile.

Da, anders als bei einem Automaten, die Struktur und das Verhalten eines Systems bei einem Petrinetz zwei unterschiedliche Aspekte darstellen, können wir die Komposition zweier Netze ausschließlich strukturell definieren. Um zwei Automaten zu komponieren, müssen wir deren Produktautomaten bilden und somit das gesamte Verhalten betrachten. Wir wollen zudem in der Regel mehr als zwei Systeme komponieren, sodass die Betrachtung als Automaten sehr komplex wird. Bei Petrinetzen können wir selbst nach der Komposition einfach über die einzelnen Komponenten argumentieren.

Wenn wir im nächsten Kapitel Transformationsregeln für Adapter einführen, sehen wir zudem, dass sich eine einzelne Transformationsregel kanonisch in eine Petrinetzstruktur einbetten lässt, was mit anderen Formalismen nur umständlich zu realisieren wäre.

2.3.1 *Struktur eines Petrinetzes*

Ein *Petrinetz* beschreibt die lokale Verarbeitung von Ressourcen. Ressourcen werden auf *Plätzen* bereit gehalten, wobei ein Platz grundsätzlich eine beliebige Anzahl nicht unterscheidbarer Ressourcen vorhalten kann. Eine *Transition* kann diese Ressourcen verarbeiten. Eine *Flussrelation* gibt an, von welchen Plätzen eine Transition Ressourcen konsumiert, und auf welchen Plätzen sie Ressourcen produziert. Die Verteilung der Ressourcen auf die Plätze nennen wir *Markierung*, eine einzelne Ressource dementsprechend *Marke*.

Definition 1 (Petrinetz [91])
Wir definieren als *Petrinetz* N das Viertupel $N = \langle P, T, F, \alpha \rangle$, wobei

- P eine endliche Menge von Plätzen ist,
- T eine endliche Menge von Transitionen ist,
- die beiden Mengen P und T disjunkt sind,
- $F : (P \times T) \cup (T \times P) \to \mathbb{N}$ die Flussrelation ist, und

- $\alpha : P \to \mathbb{N}$ die Anfangsmarkierung ist.

Im weiteren Verlauf nehmen wir implizit an, dass ein beliebiges Netz N aus den entsprechend benannten Teilen P, T, F, α besteht. Sollten wir mehrere Netze betrachten werden wir die einzelnen Teile durch einen Index kennzeichnen. Die Menge P_N repräsentiert dann zum Beispiel die Menge der Plätze des Netzes N.

Die Elemente der Menge $P \cup T$ bezeichnen wir auch als die *Knoten* eines Netzes N. Zwischen zwei Knoten $n, n' \in P \cup T$ existiert eine gerichtete *Kante*, wenn die Flussrelation für (n, n') nicht 0 ist, also $F(n, n') > 0$. Die Zahl $F(n, n')$ nennen wir das *Kantengewicht* der entsprechenden Kante.

Für einen einzelnen Knoten definieren wir auf Basis der Flussrelation den *Vor- und Nachbereich*. Für zwei Knoten $n, n' \in (P \cup T)$ liegt n' im Vorbereich von n, falls es eine Kante von n' nach n gibt. Analog liegt n' im Nachbereich von n, falls es eine Kante von n nach n' gib. Formal fasst dies folgende Definition zusammen.

Definition 2 (Vorbereich und Nachbereich)
Sei $N = \langle P, T, F, \alpha \rangle$ ein Petrinetz und $n \in P \cup T$ ein beliebiger Knoten. Der *Vorbereich* ${}^\bullet n$ von n ist definiert als ${}^\bullet n = \{n' \mid F(n', n) > 0\}$. Der *Nachbereich* $n^\bullet$ von n ist definiert als $n^\bullet = \{n' \mid F(n, n') > 0\}$.

Im Sinne dieser Definition benutzen wir die Begriffe *Vorplatz*, *Nachplatz*, *Vortransition* und *Nachtransition*, um anzudeuten, dass sich ein Platz beziehungsweise eine Transition im Vor- oder Nachbereich eines gegebenen Knotens befindet.

Ein Beispiel für ein Petrinetz sehen wir Abbildung 3. Wie für Petrinetze üblich, benutzen wir einen Kreis zur Darstellung eines Platzes, ein Rechteck für eine Transition und einen kleinen schwarzen Kreis zur Repräsentation einer Marke. Die Flussrelation wird durch Pfeile ausgedrückt. Das Kantengewicht wird durch die entsprechende Zahl auf dem Pfeil angezeigt. Steht dort keine Zahl, ist das Kantengewicht 1. Fehlt der Pfeil komplett, ist das Kantengewicht 0.

In der Abbildung sehen wir fünf Plätze $\mathrm{p}_1, \ldots, \mathrm{p}_5$ und fünf Transitionen $\mathrm{t}_1, \ldots, \mathrm{t}_5$. Das Kantengewicht zwischen Transition t_2 und Platz p_3 beträgt $F(t_2, p_3) = 3$, und zwischen Platz p_3 und Transition t_3 beträgt es $F(p_3, t_3) = 2$. Für die übrigen Kanten beträgt das Kantengewicht 1. Die Anfangsmarkierung α markiert die Plätze p_1 und p_5 und lautet somit $\alpha = [p_1, p_5]$. Der Vorbereich des Platzes p_1 ist die Transition ${}^\bullet p_1 = \{t_1\}$,

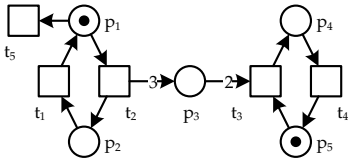


Abbildung 3: Beispiel eines Petrinetzes

der Nachbereich besteht aus den Transitionen ${p_1}^\bullet = \{t_2, t_5\}$. Analog ist der Vorbereich der Transition t_2 der Platz ${}^\bullet t_2 = \{p_1\}$, und der Nachbereich besteht aus den Plätzen ${t_2}^\bullet = \{p_2, p_3\}$.

2.3.2 *Verhalten eines Petrinetzes*

Für ein Petrinetz N ordnet eine *Markierung* $\mu : P \to \mathbb{N}$ jedem Platz seine Anzahl an Marken zu. Sie beschreibt einen Zustand von N. Die Anfangsmarkierung α definiert somit den Anfangszustand von N.

Das Schalten einer Transition ändert die Markierung und somit den Zustand eines Petrinetzes. Eine Transition t ist *aktiviert* und darf schalten, wenn jeder Platz p im Vorbereich mindestens so viele Marken enthält, wie das entsprechende Kantengewicht beträgt. Beim *Schalten* einer Transition t wird von jedem Vorplatz das entsprechende Kantengewicht an Marken entfernt und auf jeden Nachplatz das entsprechende Kantengewicht an Marken hinzugefügt. Das Entfernen von Marken bezeichnen wir als *Konsumieren* und das Hinzufügen von Marken als *Produzieren.*

Definition 3 (Schalten einer Transition)
Sei $N = \langle P, T, F, \alpha \rangle$ ein Petrinetz und μ ein beliebige Markierung von N. Eine Transition $t \in T$ ist *aktiviert* in μ, falls $\mu(p) \geq F(p,t), \forall p \in P$. Wenn wir eine in μ aktivierte Transition $t \in T$ *schalten*, dann ergibt sich eine Folgemarkierung μ' mit $\mu'(p) = \mu(p) - F(p,t) + F(t,p), \forall p \in P$.

Im Folgenden schreiben wir

- $\mu \xrightarrow{t}$, wenn die Transition t in μ aktiviert ist,
- $\mu \to$, wenn es eine Transition gibt, die in μ aktiviert ist,

- $\mu \xrightarrow{t} \mu'$, wenn das Schalten der in μ aktivierten Transition t zu μ' führt,
- $\mu \rightarrow \mu'$, wenn es eine in μ aktivierte Transition gibt, deren Schalten zu μ' führt,
- $\mu \xrightarrow{t}\!\!\!\!\!\!\rightarrow$, wenn in μ die Transition t nicht aktiviert ist, und
- $\mu \not\rightarrow$, wenn in μ keine Transition aktiviert ist.

Der Zustandsraum eine Petrinetzes N umfasst alle Markierungen, die von einer bestimmten Markierung *erreichbar* sind. Für ein Petrinetz N und eine Markierung μ ist eine andere Markierung μ' erreichbar, wenn entweder $\mu \rightarrow \mu'$ oder es eine Sequenz an Markierungen $\mu_1, \ldots, \mu_k$ gibt, sodass $\mu \rightarrow \mu_1 \rightarrow \ldots \rightarrow \mu_k \rightarrow \mu'$. Neben diesem transitiven Abschluss wollen wir die Erreichbarkeit auch reflexiv abschließen, indem wir festlegen, dass eine Markierung μ immer von sich selbst erreichbar ist. Wir schreiben $\mu \overset{\vec{t}}{\Rightarrow} \mu'$, wenn μ' über eine möglicherweise leere Sequenz $\vec{t}$ von Transitionen von μ aus erreichbar ist, oder einfach $\mu \Rightarrow \mu'$, wenn uns die Sequenz der Transitionen nicht interessiert.

Wir sagen, dass zwei Transitionen t_1 und t_2 im *Konflikt* stehen, wenn sie bezüglich einer Markierung μ beide aktiviert sind, und sich Teile ihres Vorbereichs überschneiden, also $\mu \xrightarrow{t_1}$, $\mu \xrightarrow{t_2}$ und ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$.

Als Erreichbarkeitsgraph definieren wir alle von der Anfangsmarkierung erreichbaren Markierungen zusammen mit der Relation über die direkten Nachfolger. Der Erreichbarkeitsgraph beschreibt damit das mögliche Verhalten eines Petrinetzes.

Definition 4 (Erreichbarkeitsgraph)
Sei $N = \langle P, T, F, \alpha \rangle$ ein Petrinetz. Wir definieren den *Erreichbarkeitsgraphen* $\mathfrak{R}(N)$ von N als gelabeltes Transitionssystem $\mathfrak{R}(N) = \langle Q, q_0, E \rangle$ mit

- Q ist die Menge der von der Anfangsmarkierung α erreichbaren Zustände, $Q = \{\mu \mid \alpha \Rightarrow \mu\}$,
- $q_0 \in Q$ ist der Anfangszustand, $q_0 = \alpha$, und
- $E \subseteq Q \times T_N \times Q$ sind den Zustandsübergängen entsprechende Kanten, $E = \{\langle \mu, t, \mu' \rangle \mid \mu \xrightarrow{t} \mu'\}$ mit der schaltenden Transition als Label.

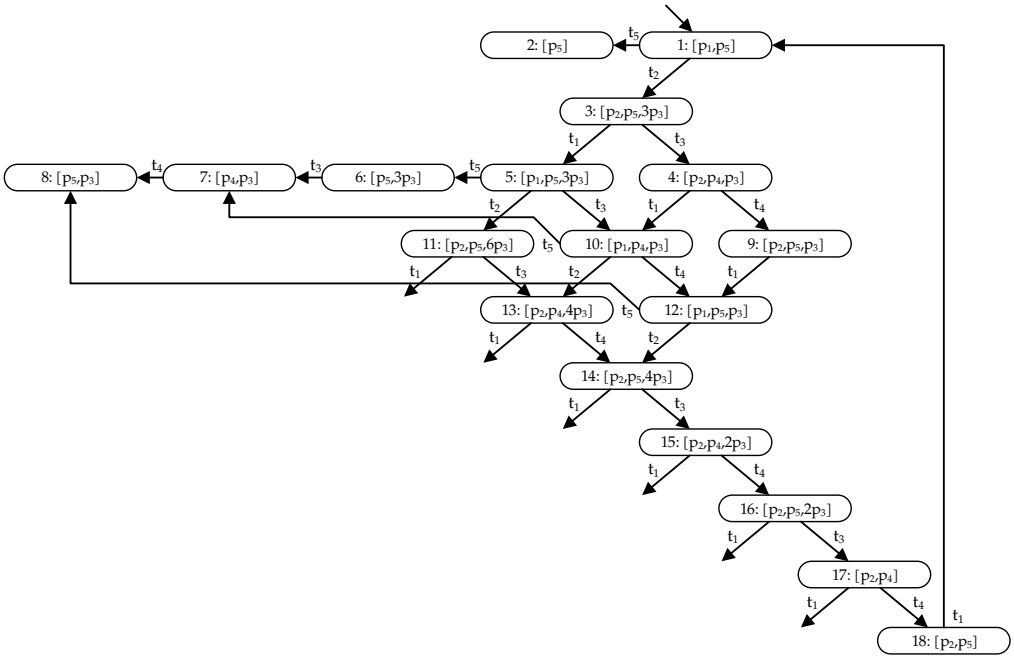


Abbildung 4: Teil des Erreichbarkeitsgraphen für das Petrinetz aus Abbildung 3

In Abbildung 4 sehen wir für das Petrinetz aus Abbildung 3 einen Teil des Erreichbarkeitsgraphen. Ein Zustand ist eine umrandete Fläche mit der entsprechenden Markierung und, zur besseren Unterscheidung, mit einer Nummer. Ein Zustandsübergang ist eine Kante mit der geschalteten Transition als Label. Der Zustand 1 ist der Anfangszustand, was wir durch den eingehenden Pfeil ohne Quelle kennzeichnen. Die Pfeile im unteren Teil der Abbildung sollen andeuten, dass hier der Erreichbarkeitsgraph weiteres Verhalten des Petrinetzes beinhaltet, dass wir nicht abbilden.

Wir sehen am Erreichbarkeitsgraphen, dass das Schalten von t_5 im Anfangszustand $[p_1, p_5]$ dazu führt, dass gar keine Transition mehr schalten kann. Solange t_5 jedoch nicht schaltet, können die Transitionen t_2 und t_1 wiederholt abwechselnd schalten. Bei jedem Schalten von t_2 werden jeweils 3 Marken auf p_3 erzeugt. Solange wenigstens 2 Marken auf p_3 liegen, können auch die Transitionen t_3 und t_4 wiederholt abwechselnd schalten.

Das Petrinetz zeigt wiederkehrendes Verhalten, das sich als Zyklus im Erreichbarkeitsgraphen äußert. Im Anfangszustand beginnend, führt zum Beispiel die Sequenz t_2, t_3, t_4, t_1, t_2, t_3, t_4, t_3, t_4, t_1 wieder zum Anfangszustand zurück.

Andererseits können sich auf dem Platz p_3 unbeschränkt viele Marken ansammeln. Wenn zum Beispiel ausschließlich die Transitionen t_2 und t_1 schalten, dann überschreitet die Anzahl der Marken auf p_3 irgendwann jede beliebige Zahl. Es gibt noch weitere Schaltsequenzen, welche dieses Verhalten verursachen.

Eine für uns wichtige Eigenschaft im Verhalten eines Petrinetzes N stellt dessen *Beschränktheit* dar. Ein Petrinetz N ist dann beschränkt, wenn es eine natürliche Zahl b gibt, sodass in keiner erreichbaren Markierung ein Platz mit mehr als b Marken existiert. Wir nennen b die *Schranke* für N. Wir werden im weiteren Verlauf der Arbeit die Beschränktheit eines Petrinetzes voraus setzen und in der Regel sogar eine Schranke b als Vorgabe betrachten.

Definition 5 (Beschränkheit)
Sei $N = \langle P, T, F, \alpha \rangle$ ein Petrinetz und $b \in \mathbb{N}$ eine natürliche Zahl. Wir nennen einen Platz $p \in P$ *b-beschränkt*, wenn in jeder von der Anfangsmarkierung α erreichbaren Markierung μ gilt, dass die Anzahl der Marken auf p kleiner gleich b ist ($\mu(p) \leq b$). Wir nennen N *b-beschränkt*, falls jeder Platz b-beschränkt ist. N ist *beschränkt*, falls ein b existiert, sodass N b-beschränkt ist.
Ist N 1-beschränkt, dann nennen wir N *sicher*.

Wenn ein Petrinetz beschränkt ist, dann ist dessen Erreichbarkeitsgraph endlich.

Lemma 6 (Endlichkeit des Erreichbarkeitsgraphen [102])
Sei $N = \langle P, T, F, \alpha \rangle$ ein Petrinetz. Der Erreichbarkeitsgraph $\mathfrak{R}(N)$ von N ist genau dann endlich, wenn N beschränkt ist.

2.3.3 *Kommunikation*

Bis hierher bilden Petrinetze geschlossene Systeme. Um mit einem Petrinetz ein offenes System modellieren zu können, fügen wir *Kommunikationslabel* an Transitionen ein und erhalten so *offene Netze*.

Die grundlegende Idee eines Kommunikationslabel für eine Transition besteht darin, anzugeben, ob eine Transition über einen Kanal c eine Nachricht empfängt, sendet oder ob sich die Transition über c synchronisiert. Die Bedeutung der Kommunikationslabel manifestiert sich in der Komposition mit einem zweiten offenen Netz als Partner in der Kommunikation.

In einem Petrinetz betrachten wir keine Daten, daher abstrahieren wir von der zu übertragenden Information und betrachten lediglich, dass eine Information über einen bestimmten Kanal übertragen wird.

Definition 7 (Kommunikationslabel)
Ein *Kommunikationslabel* ist die Bezeichnung eines Kanals c zusammen mit der Art der Kommunikation:

- $?c$ asynchrones Empfangen einer Nachricht über c
- $!c$ asynchrones Senden einer Nachricht über c
- $\#c$ Synchronisieren über c

Entsprechend dem Label bezeichnen wir einen Kanal c als *Empfangskanal* ($?c$) oder als *Sendekanal* ($!c$).

Die Schnittstelle eines offenen Netzes besteht aus der Gesamtheit aller Kommunikationslabel und deren Zuordnung zu Transitionen. Wir wollen die Schnittstelle aber noch weiter strukturieren.

Wir wählen diese Strukturierung, da wir gerade im Adaptersetting ein offenes Netz – die Engine, die wir im nächsten Kapitel einführen werden – in der Regel mit drei anderen offenen Netzen komponieren wollen. Die Komposition über Ports erlaubt, die Komposition eindeutig und unabhängig von der Reihenfolge zu gestalten.

In dieser Arbeit betrachten wir den Fall, dass die *Schnittstelle* eines offenen Netzes in *Ports* partitioniert ist. Ein Port besteht aus einer Menge von Kommunikationslabeln und dient im Folgenden als Teil der Komposition zweier offener Netze. Für eine Transition definieren wir pro Port die Zuordnung der Kommunikationslabel. Wenn eine Transition über einen Port nicht kommuniziert, benutzen wir das spezielle Label τ.

Um die Kommunikation in einem offenen Netz auszudrücken, definieren wir folgendermaßen eine Schnittstelle. Eine Schnittstelle beschreibt eine Menge von Ports, wobei jeder Port eine Menge von Kommunikationslabeln umfasst. Eine Labelfunktion λ weist für jedes Paar von Port und Transition ein Kommunikationslabel beziehungsweise τ zu, falls die Transition nicht über den Port kommuniziert.

Definition 8 (Port und Schnittstelle)
Ein *Port port* ist eine endliche, nicht leere Menge von Kommunikationslabeln.
Eine *Schnittstelle* $\mathcal{I}$ besteht aus einer endlichen Mengen an Ports.

Auch beim Synchronisieren über einen Kanal könnten wir theoretisch eine Richtung annehmen. Allerdings ist diese nur relevant, wenn wir den Inhalt einer Nachricht betrachten, da dann eine Seite den Inhalt sendet und die andere Seite den Inhalt empfängt. Die eigentliche Kommunikation findet gleichzeitig statt und hat damit keine ausgewiesenen Initiator. Da wir vom Inhalt einer Nachricht abstrahieren, ist ein synchroner Kanal somit ungerichtet.

Zusammen mit der Kommunikation betrachten wir bei einem offenen Netz auch dessen Terminierung, sodass zur Schnittstelle und deren Abbildung auf die Transitionen eine Menge von *Endmarkierungen* hinzukommt. Dabei drückt eine Endmarkierung aus, dass ein offenes Netz in solch einer Markierung terminieren darf. Es muss jedoch nicht terminieren, sondern der Ablauf kann weiter gehen, wobei die endgültige Entscheidung zur Terminierung der Umgebung des Systems überlassen wird. Dies ist sinnvoll, wenn ein Kommunikationspartner zur gleichen Zeit noch keine Endmarkierung erreicht hat, und somit das offene Netz weiter zur Interaktion zur Verfügung stehen sollte.

Definition 9 (Offenes Netz [48])
Wir definieren ein *offenes Netz* N als 7-Tupel $N = \langle P, T, F, \alpha, \Omega, \mathcal{I}, \lambda \rangle$ mit

- $\langle P, T, F, \alpha \rangle$ ist ein Petrinetz,
- Ω ist eine endliche Menge von Endmarkierungen über P, also Markierungen, in denen N terminieren darf,
- $\mathcal{I}$ ist die Schnittstelle von N, und

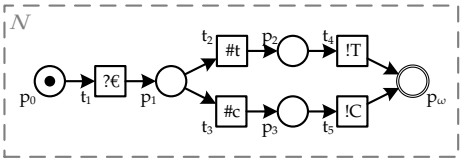


Abbildung 5: Beispiel eines offenen Netzes N

- $\lambda : \mathcal{I} \times T \to port \cup \{\tau\}$ ordnet für jeden Port $port \in \mathcal{I}$ einer Transition $t \in T$ ein Kommunikationslabel des entsprechenden Ports oder $\tau \notin port$ zu, wobei τ bedeutet, dass t über $port$ nicht kommuniziert.

Andere Arbeiten [61, 114] haben die Kommunikation mit Hilfe von zusätzlichen Plätzen eingeführt, sodass eine kommunizierende Transition nur schalten kann, wenn bestimmte Bedingungen an die Kommunikationsplätze erfüllt sind. Darauf verzichten wir an dieser Stelle. Die Schaltregel für ein offenes Netz gilt unabhängig von den Labeln der Transitionen. Wann immer wir das Verhalten eines offenen Netzes unter Berücksichtigung der Kommunikation betrachten wollen, dann wird dieses Netz entsprechend mit einem weiteren offenen Netz komponiert sein.

Wir betrachten in Abbildung 5 das Beispiel eines Getränkeautomaten, der dort als offenes Netz modelliert ist. Die Kommunikationslabel stehen in ihrer zugehörigen Transition. Der Platz, der in der Endmarkierung markiert sein soll, ist doppelt umrandet. Das gesamte offene Netz ist von einer gestrichelten Box umgeben um anzuzeigen, dass das Netz dort endet. Dies dient der Deutlichkeit, wenn wir mehrere offene Netze in einer Abbildung zeigen.

Das Netz empfängt am Anfang asynchron einen Geldbetrag über den Kanal €. Anschließend schaltet entweder Transition t_2 und synchronisiert sich über den Kanal t, der Knopfdruck für Tee, oder es schaltet Transition t_3 mit einer Synchronisation über Kanal c, der Knopfdruck für Cola. Im ersten Fall wird danach asynchron der Tee über den Kanal T gesendet. Im zweiten Fall wird asynchron die Cola über den Kanal C gesendet. In beiden Fällen ist dann eine Endmarkierung erreicht und keine weiter Transition aktiviert.

2.3.4 *Komposition offener Netze*

Die Bedeutung der Kommunikation eines offenes Netzes ergibt sich durch die Komposition mit einem zweiten offenen Netz. Uns interessiert die Komposition mehrerer Netze, sodass sich ein geschlossenes System bildet.

Ein offenes Netz beschreibt ein geschlossenes System, wenn keine Transitionen mehr mit einem Kommunikationslabel außer τ beschriftet ist. Für ein geschlossenes System lassen sich sinnvoll Aussagen über das auftretende Kommunikationsverhalten zwischen offenen Netzen treffen. Verbleibt im Gegensatz dazu ein Kommunikationslabel im offenen Netz, dann ist die Kommunikation für die entsprechende Transition nicht vollkommen bestimmt. Über einen uns unbekannten Kommunikationspartner können wir keine – oder zumindest deutlich weniger – verlässliche Aussagen treffen.

Zwei offene Netze komponieren wir nur dann, wenn jedes Netz einen Port besitzt, den wir sinnvoll mit dem Port des jeweils anderen Netzes verknüpfen können. Die beiden Ports dürfen dafür weder gleiche Empfangs- noch Sendekanäle enthalten.

Definition 10 (Komponierbare Ports)
Seien $port_1$ und $port_2$ zwei Ports. Wir nennen $port_1$ und $port_2$ *komponierbar*, wenn

- $?c \in port_1 \iff !c \in port_2$,
- $!c \in port_1 \iff ?c \in port_2$ und
- $\#c \in port_1 \iff \#c \in port_2$.

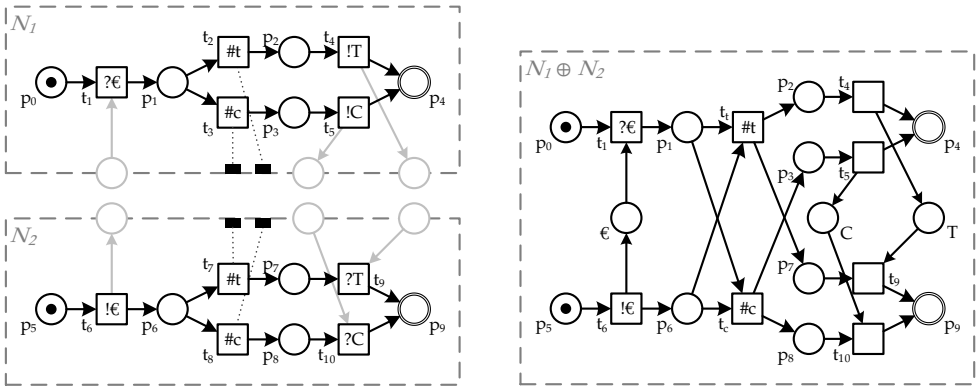
Bei der Komposition zweier offenen Netz entsteht ein neues offenes Netz, das die Semantik hinter den Kommunikationslabeln, wie oben angegeben, umsetzt. Wir gehen davon aus, dass die Mengen der Plätze und Transitionen zweier zu komponierender offener Netze jeweils disjunkt sind, und dass kein Kommunikationslabel wie ein Platz oder Transition heißt. Falls diese Eigenschaft nicht gilt, können wir sie durch Umbenennen sicher stellen. In der Komposition entsteht ein Kommunikationsplatz für jedes asynchrone Label und für jedes synchrones Label benennen wir die entsprechende Transition in den Namen des Labels um. Damit ist Komposition nach diesem Schritt die Vereinigung der beiden offenen Netze.

Definition 11 (Komposition offener Netze)
Seinen $N_1 = \langle P_1, T_1, F_1, \alpha_1, \Omega_1, \mathcal{I}_1, \lambda_1 \rangle$ und $N_2 = \langle P_2, T_2, F_2, \alpha_2, \Omega_2, \mathcal{I}_2, \lambda_2 \rangle$ zwei offene Netze. Wir definieren als *Komposition* $N = N_1.port_1 \oplus N_2.port_2$ an den komponierbaren Ports $port_1$ und $port_2$ das Netz $N = \langle P, T, F, \alpha, \Omega, \mathcal{I}, \lambda \rangle$ wie folgt:

- $P = P_1 \cup P_2 \cup P_c$ wobei
  $P_c = \{c \mid ?c \in port_1 \text{ oder } !c \in port_1\}$ eine Menge von *Kommunikationsplätzen* ist,
- $T = T_1' \cup T_2'$ wobei für $i = 1, 2$ T_i' sich aus T_i ergibt, indem $t \in T_i$ mit $\lambda_i(port_i, t) = \#c$ in t_c umbenannt wird, d. h., t_c den Platz von t einnimmt mit dem entsprechenden Vor- und Nachbereich,
- für $i = 1, 2$
  $$F(n, n') = \begin{cases} F_1 & \text{falls } n, n' \in P_1 \cup T_1 \\ F_2 & \text{falls } n, n' \in P_2 \cup T_2 \end{cases}$$
- $\alpha = \alpha_1 + \alpha_2$,
- $\Omega = \{\omega_1 + \omega_2 \mid \omega_1 \in \Omega_1, \omega_2 \in \Omega_2\}$,
- $\mathcal{I} = (\mathcal{I}_1 \cup \mathcal{I}_2) \setminus \{port_1, port_2\}$, und
- für $port \in \mathcal{I}$, $t \in T$ und $i = 1, 2$
  $$\lambda(port, t) = \begin{cases} \lambda_i(port, t) & port \in \mathcal{I}_i, t \in T_i \\ \lambda_i(port, t_i) & port \in \mathcal{I}_i, t = (t_1 t_2) \in T_c, t_i \in T_i \\ \tau & \text{sonst} \end{cases}$$

Die Addition auf Markierungen $\mu = \mu_1 + \mu_2$ sei dabei wie folgt platzweise definiert:

$$\mu(p \in P) = \begin{cases} \mu_1(p) & p \in P_1 \\ \mu_2(p) & p \in P_2 \\ 0 & p \in P_c \end{cases}$$



(a) Vor der Komposition

(b) Nach der Komposition

Abbildung 6: Komposition der offenen Netze N_1 und N_2

NOTATION: Sollte aus dem Zusammenhang klar sein, welche Ports wir komponieren wollen, beziehungsweise dies unerheblich sein, werden wir aus Gründen der Lesbarkeit auf deren Angaben verzichten und kurz $N_1 \oplus N_2$ schreiben.

Eine Veranschaulichung der Komposition zweier offener auf Basis des bereits gezeigten Getränkeautomaten sehen wir in Abbildung 6. Wir komponieren die beiden offenen Netze N_1 und N_2 in Abbildung 6a und sehen das Ergebnis der Komposition in Abbildung 6b.

Das offene Netz N_1 entspricht dem Getränkeautomaten aus Abbildung 5. Das offene Netz N_2 sollen einen Kunden darstellen, der € sendet, eine Taste drückt und das entsprechende Getränk empfängt. Für beide Netze nehmen wir jeweils nur einen Port an. Um das Verständnis der beiden offenen Netz zu erleichtern und den Effekt der Komposition vorwegzunehmen, sind die entsprechenden Kommunikationsplätze in grau angedeutet und die schwarzen Kästchen, die mit einer gepunkteten Linie mit einer Transition verbunden sind, zeigen an, welche Transitionen bei der Komposition vereinigt werden.

Das Ergebnis der Komposition $N_1.port_1 \oplus N_2.port_2$ sehen wir in Abbildung 6b. Für die asynchronen Kanäle existieren entsprechende Plätze €, C und T. Die Transitionen, die über diese Kanäle senden oder empfangen sollen, haben entsprechende Kanten. So produziert nun Transition t_6, die in Abbildung 6a das Label $\lambda(port_2, t_6) = !€$ hat, auf den

Platz €. Für jedes Paar von Transitionen in N_1 und N_2 mit dem gleichen synchronen Kommunikationslabel gibt es eine Transition in $N_1.port_1 \oplus N_2.port_2$. Für das Paar t_2 und t_7 enthält $N_1.port_1 \oplus N_2.port_2$ die Transition t_t, da die beiden Transitionen bei der Komposition in t_t umbenannt wurden. Diese Transition vereinigt entsprechend den Vor- und Nachbereich der Transitionen t_2 und t_7 in sich.

2.3.5 *Korrektheit eines offenen Netzes*

Bei der Modellierung eines Systems spielt dessen Korrektheit eine wichtige Rolle. Wann ein System korrekt ist, hängt jedoch stark vom Zweck dieses Systems ab.

Viele praktisch relevante Korrektheitskriterien lassen sich auf Verklemmungsfreiheit, Lebendigkeits, Beschränktheit und so weiter reduzieren. Zusätzlich betrachten wir bei einem offenen Netz noch Endmarkierungen, um über Terminierung argumentieren zu können.

Eine der grundlegendsten Anforderungen an ein offenes Netz N ist, dass es nur in einer Endmarkierung terminieren darf, worauf wir in Definition 9 eines offenen Netzes bereits hinweisen. Das heißt, N ist nur dann gut, wenn jede erreichbare Markierung in N eine Transition aktiviert oder eine Endmarkierung ist. Diese Anforderung ist eine Variante der *Verklemmungsfreiheit* für ein Petrinetz unter Berücksichtigung von Endmarkierungen. Ist die Menge der Endmarkierungen leer, erhalten wir genau die Verklemmungsfreiheit von Petrinetzen.

Ein stärkere Anforderung ist die Erreichbarkeit einer Endmarkierung – genauer: von jeder erreichbaren Markierung soll eine Endmarkierung erreichbar sein. Diese Anforderung bezeichnen wir als *Schwache Terminierung*. Gerade im Umfeld von Webservices oder Systemen mit transaktionalem Charakter möchten wir sicher sein, dass ein System die Möglichkeit hat, seine Aufgaben abzuschließen.

Wenn wir nicht nur die Möglichkeit der Terminierung verlangen, sondern auch, dass ein System grundsätzlich terminiert, dann sprechen wir von *Starker Terminierung*. Das heißt, obwohl in solch einem System auch unendliches Verhalten möglich ist, brauchen wir ein Zusicherung, dass das System nach endlich vielen Schritten terminiert.

Die obigen Kriterien lassen sich gut auf geschlossene Systeme anwenden. Sie sind dabei Grundlage weiterer Eigenschaften wie zum Beispiel *Soundness* [4], einer Kombination aus Lebendigkeit und Beschränkheit.

Bei einem offenen System hingegen müssen wir einen Partner des offenen Systems in Betracht ziehen, um über dessen Korrektheit aussagen zu treffen. Die Kommunikation hat entscheidenden Einfluss auf das Verhalten eines offenen System. Ein offenes Netz N ist also dann korrekt, wenn es in Komposition mit einem N' korrekt ist. In dieser Arbeit bezeichnen wir solch ein N' als *Controller* für N, wenn es dessen Korrektheit sicher stellt. Für die obigen Kriterien können wir auf Vorarbeiten [61, 112, 115] zurückgreifen, die sich mit der Interaktion offener Netze beschäftigen. In den entsprechenden Arbeiten wird unter anderem jeweils ein allgemeinster Partner definiert, den wir in dieser Arbeit als Controller nutzen.

Allgemein funktioniert der in dieser Arbeit vorgestellte Ansatz, wenn wir für ein bestimmtes Korrektheitskriterium einen *Controller* für ein offenes Netz generieren können, beziehungsweise die Existenz eines solchen ausschließen können. In der Anwendung der Technik benutzen wir jedoch insbesondere den *allgemeinsten Kommunikationspartner*, sodass sich die Resultate allgemein auf Korrektheitskriterien übertragen lassen, für die ein allgemeinster Kommunikationspartner berechnet werden kann.

2.4 CONTROLLERSYNTHESE

Eine wichtige Frage bei der Modellierung eines offenen Systems ist die nach der Existenz eines sinnvollen Kommunikationspartners. Gegeben ein offenes Netz N und ein Korrektheitskriterium, existiert ein zweites Netz N', sodass die Komposition $N \oplus N'$ das Korrektheitskriterium erfüllt? Da diese Frage eng mit der Frage verbunden ist, ob wir N so steuern können, dass es korrekt ist, bettet sich die Frage in den Bereich der Controllersynthese ein. Ein gegebenes System zusammen mit dem Controller soll sich korrekt verhalten.

Definition 12 (Controller)
Seien N_1 und N_2 zwei offene Netze. Dann nennen wir N_2 einen *Controller* für N_1, wenn die Komposition $N_1 \oplus N_2$ korrekt ist.

Wir nennen N_1 *kontrollierbar*, wenn ein Controller N_2 für N_1 existiert.

Die Art der Controllersynthese hängt vom Formalismus ab, in dem ein System modelliert ist. Im Fall der offenen Netze gibt es verschieden Vorarbeiten, um Korrekt-

heitskriterien wie Verklemmungsfreiheit [64], Verklemmungsfreiheit mit garantierter Kommunikation [57] oder Schwache Terminierung [112] nutzen zu können.

Eine Besonderheit der oben genannten Korrektheitskriterien ist die Existenz eines *allgemeinsten* Kommunikationspartners M, eines speziellen Controllers, für ein gegebenes offenes Netz N. Solche ein allgemeinster Kommunikationspartner erlaubt das maximale Kommunikationsverhalten in N, das heißt, jeder weitere Kommunikationspartner N' zeigt in Komposition $N \oplus N'$ maximal soviel Kommunikationsverhalten wie die Komposition $N \oplus M$.

Eine Voraussetzung für die Existenz eines allgemeinsten Kommunikationspartners in den genannten Fällen ist jedoch die Annahme, dass ein gegebenes offenes Netz N beschränkt ist, also dessen Erreichbarkeitsgraph endlich, und dass in Komposition auf den eingefügten Kommunikationsplätzen eine vorgegebene Schranke nicht überschritten wird. Die Menge P_c aus Definition 11 über die Komposition offener Netze umfasst eben jene Kommunikationsplätze.

Definition 13 (Kommunikationsschranke)
Seien N_1 und N_2 zwei offene Netze mit komponierbaren Ports. Wir definieren eine *Kommunikationsschranke* $k \in \mathbb{N}$, für die gilt, dass die Komposition $N_1 \oplus N_2$ die Kommunikationsschranke k respektiert, wenn in jeder erreichbaren Markierung μ von $N_1 \oplus N_2$ gilt, dass auf den Kommunikationsplätzen maximal k Marken liegen, also $\mu(p) \leq k, \forall p \in P_c$.

Im Folgenden führen wir die benötigten Definitionen ein, um einen allgemeinsten Kommunikationspartner zu beschreiben. Die Definitionen sind konstruktiv, das heißt im Falle der Existenz erhalten wir einen allgemeinsten Kommunikationspartner, der bezüglich der Definitionen eindeutig ist. Sollte solch ein allgemeinster Kommunikationspartner nicht existieren, bedeutet dies, dass das gegebene offene Netz keinen Kommunikationspartner hat, mit dem das Korrektheitskriterium erfüllt werden kann.

Um einen Controller für ein offenes Netz N zu generieren, überapproximieren wir alles mögliche Verhalten, das ein Controller mit N zusammen zeigen kann. Dazu führen wir eine universelle Umgebung U_N von N ein, in der beliebiges Kommunikationsverhalten möglich ist. Wir betrachten dann das mögliche Verhalten der Transitionen von U_N in der Komposition $N \oplus U_N$. So beschreiben wir, welches Wissen U_N in einem seiner Zustände über die möglichen Markierung von $N \oplus U_N$ hat. Das Verhalten dieser

Transition beschreiben wir als Transitionssystem, in dem wir iterativ jeden Zustand entfernen, der gegen das gewünschte Korrektheitskriterium verstößt. Wenn wenigstens ein Zustand übrig bleibt, nennen wir das verbleibende Transitionssystem den allgemeinsten Kommunikationspartner beziehungsweise einen Controller von N.

Zuerst definieren wir die *universelle Umgebung*. Dies ist ein Kommunikationspartner, der jegliches Kommunikationsverhalten überapproximiert – nicht nur das korrekte.

Definition 14 (Universelle Umgebung)
Sei $N = \langle P, T, F, \alpha, \Omega, \mathcal{I}, \lambda \rangle$ ein offenes Netz mit einem einzigem Port $port$ ($\mathcal{I} = \{port\}$). Wie definieren als $\overline{port}$ das kanonische Gegenstück zu $port$, d. h., $?c \in port \implies !c \in \overline{port}$, $!c \in port \implies ?c \in \overline{port}$ und $\#c \in port \implies \#c \in \overline{port}$. Wir definieren dann als *universelle Umgebung* das offene Netz $U_N = \langle P_U, T_U, F_U, \alpha_U, \Omega_U, \mathcal{I}_U, \lambda_U \rangle$ mit

- $P_U = \emptyset$,
- $T_U = \{t_l \mid l \in \overline{port}\}$,
- $F_U = \emptyset$,
- $\alpha_U = []$,
- $\Omega_U = \{[]\}$,
- $\mathcal{I} = \{\overline{port}\}$, und
- $\lambda(t_l) = l$.

Abbildung 7 zeigt ein Beispiel für eine universelle Umgebung. Gegeben sei der Getränkeautomat als offenes Netz N in Abbildung 7a. Die universelle Umgebung U_N in Abbildung 7b hat die gleichen Kanäle wie N, bis auf dass das Senden und Empfangen jeweils vertauscht ist. Für jeden Kanal gibt es genau eine Transition, die immer aktiviert ist. In Komposition $N \oplus U_N$ bedeutet das, dass die Transitionen immer schalten können, es sei denn sie benötigen eine Marke auf dem entsprechenden Kommunikationsplatz.

Da die universelle Umgebung jegliches Kommunikationsverhalten überapproximiert, können wir sie nutzen, um das Verhalten des allgemeinsten Kommunikationspartners zu beschreiben. Das Kommunikationsverhalten von U_N mit N erhalten wir, indem wir

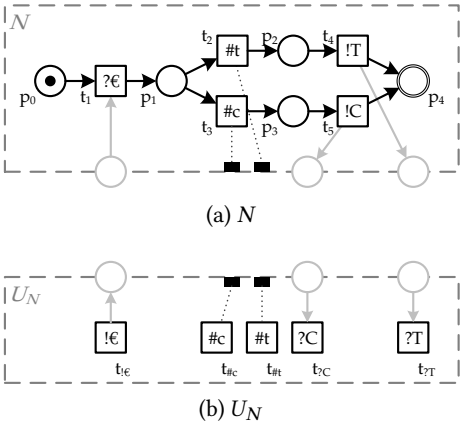


Abbildung 7: Ein offenes Netz N mit seiner universellen Umgebung U_N

beide offene Netze komponieren und dessen gemeinsamen Erreichbarkeitsgraphen in Bezug auf die Transitionen aus U_N betrachten. Wir interessieren uns also für das Verhalten einer Teilmenge der Transitionen eines offenen Netzes im Erreichbarkeitsgraphen.

Wir abstrahieren das Verhalten, indem wir Transitionen, deren Verhalten uns nicht interessiert, mit τ beschriften. Im Fall der universellen Umgebung beschriften wir alle Transitionen von N mit τ, da uns nur das Verhalten der Transitionen von U_N interessiert.

Den reflexiven und transitiven Abschluss über die erreichbaren Markierungen, wenn nur τ-Transitionen schalten, bezeichnen wir als *closure*. Die closure für eine Menge von Markierungen gibt uns also an, welche Markierungen erreichbar sind, ohne dass eine für uns beobachtbare Transition geschaltet hat.

Um bei der Benennung der offenen Netz keine Verwirrung zu stiften, beziehen sich die nachfolgenden Definitionen auf ein beliebiges offenes Netz O. Im Algorithmus zur Controllersynthese ist $O = N \oplus U_N$ und die Menge der beobachtbaren Transitionen ist T_U.

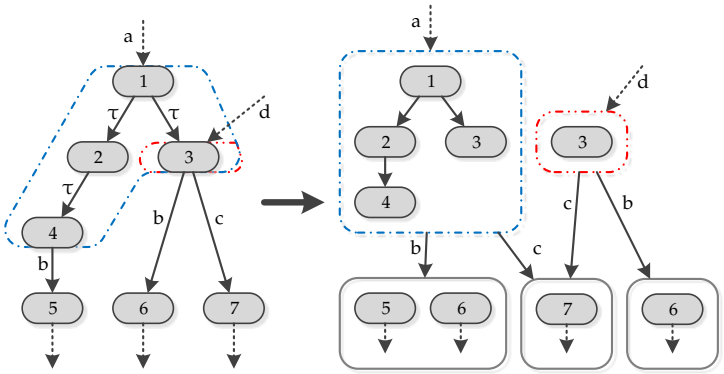


Abbildung 8: Effekt des τ-Abschlusses, von den mit τ gelabelten Transitionen abstrahieren wir und bilden Zustände mit Hilfe des τ-Abschlusses closure

Definition 15 (τ-closure)
Sei $O = \langle P, T, F, \alpha, \Omega, \mathcal{I}, \lambda \rangle$ ein offenes Netz, $T' \subseteq T$ ein Menge von beobachtbaren Transitionen und $T_\tau = T \setminus T'$ die Menge der τ-Transitionen. Für eine Menge M von Markierungen definieren wir als *τ-Abschluss closure(M)* die kleinste Menge, die folgende Eigenschaften erfüllt:

- $M \subseteq \text{closure}(M)$ und
- für $\mu \in \text{closure}(M)$ und $\mu \xrightarrow{t} \mu'$, $t \in T_\tau$ gilt $\mu' \in \text{closure}(M)$.

Der τ-Abschluss über einer Menge von Markierungen ist eindeutig definiert.

In Abbildung 8 sehen wir einen Teil eines Erreichbarkeitsgraphen. Von den mit τ gelabelten Kanten abstrahieren wir.

Betrachten wir zum Beispiel Zustand 1 auf der linken Seite. Von diesem erreichen wir über τ-Transitionen die Zustände 2, 4 und 3. Damit ist der Abschluss der Menge $\{1\}$ die Menge $\text{closure}(\{1\}) = \{1, 2, 3, 4\}$. Da vom Zustand 3 aus keine Zustände über τ-Transitionen erreichbar sind, ist der Abschluss $\text{closure}(\{3\}) = \{3\}$.

Im allgemeinsten Kommunikationspartner bilden diese Abschlussmengen die Zustände. Ein Zustandsübergang findet durch das Schalten einer der beobachtbaren Transitionen statt.

Definition 16 (Schritt auf einer Markierungsmenge)
Sei $O = \langle P, T, F, \alpha, \Omega, \mathcal{I}, \lambda \rangle$ ein offenes Netz, $T' \subseteq T$ ein Menge von beobachtbaren Transitionen und M eine Menge von Markierungen in O. Eine Transition $t \in T'$ ist *in M möglich*, wenn es im Erreichbarkeitsgraphen von O in μ einen Übergang mit t gibt. Wenn eine Transition $t \in T'$ in M möglich ist, dann erhalten wir durch einen *Schritt step* in M mit t folgende Menge:

$$\text{step}(M, t) = \{\mu' \mid \mu \in M, \mu \xrightarrow{t} \mu'\}$$

Die Erweiterung auf Kommunikationslabel umfasst dann alle Transitionen, die bezüglich eines Ports *port* ein entsprechendes Label l besitzen:

$$\text{step}(M, l) = \{\mu' \mid \mu \in M, \mu \xrightarrow{t} \mu', \lambda(port, t) = l\}$$

Mit den beiden Definitionen für Abschluss und Schritt, können wir nun ein Transitionssystem auf eine bestimmte Menge von Transitionen abstrahieren.

Definition 17 (Beobachtbares Verhalten)
Sei $O = \langle P, T, F, \alpha, \Omega, \mathcal{I}, \lambda \rangle$ ein offenes Netz und $T' \subseteq T$ ein Menge von beobachtbaren Transitionen. Dann definieren wir das *beobachtbare Verhalten von O bezüglich T'* rekursiv als Transitionssystem $\mathcal{B}(O, T') = \langle Q, q_0, E \rangle$ mit

- $q_0 = \text{closure}(\alpha) \in Q$,
- wenn $q \in Q$ und $t \in T'$ möglich in q,
  dann $q' = \text{closure}(\text{step}(q, t)) \in Q$ und $\langle q, t, q' \rangle \in E$.

Die Definition erweitert sich kanonisch auf eine Menge von Kommunikationslabeln mit der entsprechenden Schrittsemantik.

In Abbildung 7 haben wir den Getränkeautomaten N mit seiner universellen Umgebung U_N betrachtet. Das Verhalten von $O = N \oplus U_N$ sehen wir ausschnittsweise in Abbildung 9.

Innerhalb der Zustände sehen wir die Markierungsmengen von $N \oplus U_N$, welche ohne das Schalten einer Transition von U_N möglich ist. Im Anfangszustand befindet

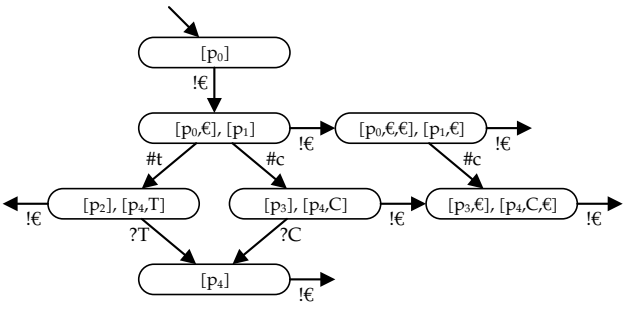


Abbildung 9: Ausschnitt aus dem Verhalten von $N \oplus U_N$ aus Abbildung 7 abstrahiert auf die Transitionen von U_N.

sich $N \oplus U_N$ in der Anfangsmarkierung $[p_0]$. Im Erreichbarkeitsgraphen ist in dieser Markierung nur der Übergang mit der Transition !€ möglich, der sich entsprechend auch beim Anfangszustand wiederfindet. Dieser Übergang führt zur Markierung $[p_0, €]$, die über eine τ-Transition die Markierung $[p_1]$ in $N \oplus U_N$ erreicht. Daher sind diese beiden Markierungen im Folgezustand enthalten. In beiden Markierungen ist der Übergang !€ möglich, der nach rechts weiter führt. Aus der Markierung $[p_0, €]$ ist der Übergang #t möglich. Wir erreichen damit die Markierung $[p_2]$ und über eine weitere τ-Transition die Markierung $[p_4, T]$, die beide im entsprechenden Zustand enthalten sind. Das Schalten von !€ ist in jeder Markierung und somit in jedem Zustand möglich. Somit ist sowohl der Erreichbarkeitsgraph als auch das abstrahierte Verhalten unendlich. Allerdings verletzt der Zustand $[p_0, €, €], [p_1, €]$ die Kommunikationsschranke – wenn wir sie mit 1 annehmen –, weil es eine Markierung mit mehr als einer Marke auf dem Platz € gibt. Im Folgenden beschränken wir das Transitionssystem, indem wir Übergänge von Zuständen, die die Kommunikationsschranke verletzten, nicht verfolgen.

Wir haben nun alle Definitionen beisammen, die es uns erlauben, den allgemeinsten Kommunikationspartner eines offenen Netzes N zu synthetisieren. Wir betrachten die Komposition $N \oplus U_N$ von N mit seiner universellen Umgebung U_N, bilden deren Erreichbarkeitsgraphen und abstrahieren diesen bezüglich der Transitionen von U_N. Der Graph $\mathcal{B}(N \oplus U_N, T_U)$ beschreibt dann das maximal mögliche Kommunikationsverhalten mit N. Nun müssen wir die Korrektheit sicherstellen.

In dieser Arbeit werden wir Schwache Terminierung als zentrales Korrektheitskriterium betrachten. Im Falle eines Adapters, wie wir ihn im kommenden Kapitel definieren, soll dieser nicht isoliert arbeiten, während die zu adaptierenden Systeme untätig sind. Daher ist Schwache Terminierung hier das interessantere Korrektheitskriterium, da der Adapter sicherstellen muss, dass die zu adaptierenden Systeme einen Endzustand erreichen können. Im folgenden Abschnitt führen wir am Beispiel der Schwachen Terminierung die Controllersynthese ein.

Definition 18 (Schwache Terminierung)
Sei $O = \langle P, T, F, \alpha, \Omega, \mathcal{I}, \lambda \rangle$ ein offenes Netz. Wir nennen O genau dann *schwach terminierend*, wenn von jeder erreichbaren Markierung in O eine Endmarkierung erreichbar ist: $\alpha \Rightarrow \mu \implies \exists \omega \in \Omega, \mu \Rightarrow \omega$.

Im Fall der Schwachen Terminierung gab es zwei Eigenschaften, die in einen komponierten offenen Netz gelten müssen: In jeder erreichbaren Markierung muss eine Endmarkierung erreichbar sein, und auf einem Kommunikationsplatz dürfen niemals mehr Marken liegen als durch eine entsprechende Kommunikationsschranke vorgegeben. Wenn ein Zustand $q \in Q$ von $\mathcal{B}(N \oplus U_N, T_U)$ eine dieser Eigenschaften verletzt, dann entfernen wir diesen.

Durch Entfernen von Zuständen in $\mathcal{B}(N \oplus U_N, T_U)$ schränken wir das Verhalten ein, das ein Kommunikationspartner zeigen darf. Einen solchen Zustand zu behalten bedeutete, dass ein Kommunikationspartner inkorrektes Verhalten erreichen kann. Durch Entfernen des Zustandes und der Schritte zu diesem Zustand wird inkorrektes Verhalten aktiv vermieden.

Definition 19 (Allgemeinster Kommunikationspartner)
Sei $N = \langle P, T, F, \alpha, \Omega, \mathcal{I}, \lambda \rangle$ ein offenes Netz, $k \in \mathbb{N}$ eine Kommunikationsschranke und $U_N = \langle P_U, T_U, F_U, \alpha_U, \Omega_U, \mathcal{I}_U, \lambda_U \rangle$ die universelle Umgebung von N. Wir nehmen ein Korrektheitskriterium mit Kommunikationsschranke als gegeben an.

Sei $TS_0 = \mathcal{B}(N \oplus U_N, T_U)$ das allgemeinste Kommunikationsverhalten mit N. Wir iterieren wie folgt:

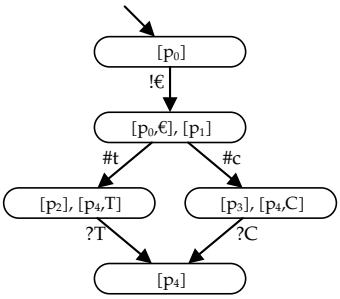


Abbildung 10: Allgemeinster Kommunikationspartner des Getränkeautomaten aus Abbildung 5.

1. Wir erhalten TS_1 aus TS_0, indem wir jeden Zustand entfernen, in dem die Kommunikationsschranke auf den Kommunikationsplätzen verletzt ist, und alle Zustände, die dadurch nicht länger erreichbar sind.

2. Wir erhalten TS_{i+1} aus TS_i, indem wir jeden Zustand entfernen, in dem das Korrektheitskriterium verletzt ist, und alle Zustände, die dadurch nicht länger erreichbar sind.

Sei $C = TS_i$ für das kleinste i, für das $TS_i = TS_{i+1}$ gilt. Wir nennen C den *allgemeinsten Kommunikationspartner* von N, sofern C wenigstens den Anfangszustand enthält.

Der mit dieser Definition berechnete Kommunikationspartner ist tatsächlich ein Controller. Dieses Ergebnis nutzen wir im nächsten Kapitel zur Adaptersynthese, wo wir eine Controller berechnen müssen.

Theorem 20 (Allg. Kommunikationspartner ist Controller [61, 115])
Sei C der allgemeinste Kommunikationspartner von N, dann erfüllt $N \oplus C$ das gegebene Korrektheitskriterium.

Den allgemeinsten Kommunikationspartner des Getränkeautomaten aus Abbildung 5 sehen wir in Abbildung 10. Es ist genau der Teil von $\mathcal{B}(N \oplus U_N, T_U)$ der übrig bleibt, wenn kein zweiter Euro gesendet wird.

Für die obige Definition wurde in früheren Arbeiten bereits für Verklemmungsfreiheit [64], Verklemmungsfreiheit mit garantierter Kommunikation [57] und schwache Terminierung [112] gezeigt, dass der allgemeinste Partner genau dann existiert, wenn

es wenigstens einen Kommunikationspartner gibt, der in Komposition mit N das entsprechende Korrektheitskriterium erfüllt.

Dass das gefundene C der *allgemeinste* Kommunikationspartner von N ist, bedeutet, dass er jeden anderen Kommunikationspartner simuliert. Simulation zwischen zwei Transitionssystemen bedeutet dabei, dass jedes Verhalten, welches das erste System zeigt, auch das zweite System zeigen kann. Wir nehmen an, dass das zweite System keine τ-Transitionen enthält. Das ist im allgemeinsten Kommunikationspartner gegeben, weil wir bei der Konstruktion gerade die Nicht-τ-Transitionen als Zustandsübergänge betrachten.

Definition 21 (Simulationsbeziehung)
Seien $TS_1 = \left\langle Q_1, q_{0_1}, E_1 \right\rangle$ und $TS_2 = \left\langle Q_2, q_{0_2}, E_2 \right\rangle$ zwei Transitionssysteme, wobei TS_2 keine τ-Transitionen enthält. TS_2 *simuliert* TS_1, wenn es eine Relation $S \subseteq Q_1 \times Q_2$ gibt, sodass

1. $\left\langle q_{0_1}, q_{0_2} \right\rangle \in S$,
2. $\langle q_1, q_2 \rangle \in S$ und $\left\langle q_1, t, q_1' \right\rangle \in E_1$ mit $t \neq \tau$ impliziert $\exists q_2' : \left\langle q_2, t, q_2' \right\rangle \in E_2$ und $\left\langle q_1', q_2' \right\rangle \in S$, und
3. $\langle q_1, q_2 \rangle \in S$ und $\left\langle q_1, \tau, q_1' \right\rangle \in E_1$ impliziert $\left\langle q_1', q_2 \right\rangle \in S$.

2.4.1 *Übersetzung eines Transitionssystems in ein Petrinetz*

Wir sind in der Lage, zu einem gegebenen offenen Netz einen Controller in Form des allgemeinsten Kommunikationspartners auszurechnen. Allerdings liegt dieser Controller in Form eines Transitionssystems vor. Wir können jedoch jedes Transitionssystem mit Kommunikationslabeln an den Kanten zurück in ein offenes Netz übersetzen.

Die einfachste Möglichkeit ist, das Transitionssystem in eine Zustandsmaschine zu übersetzen. Eine Zustandsmaschine ist eine spezielle Form von Petrinetz, in der jede Transition genau einen Vor- und einen Nachplatz besitzt. Die Struktur ähnelt daher stark einem Transitionssystem.

Definition 22 (Transitionssystem als offenes Netz)
Sei $TS = \langle Q, q_0, E \rangle$ ein Transitionssystem. Dann definieren wir das offene Netz $N_{TS} = \langle P, T, F, \alpha, \Omega, \mathcal{I}, \lambda \rangle$ wie folgt:

- $P = Q$,
- $T = E$,
- $F(q_1, t) = F(t, q_2) = 1$ für $t = \langle q_1, l, q_2 \rangle \in E$,
- $\alpha = [q_0]$,
- $\Omega = \{[p] \mid p \in P\}$,
- $\mathcal{I} = \{port\}$ mit $port = \{l \mid \langle q_1, l, q_2 \rangle \in E\}$, und
- $\lambda(port, t) = l$ für $t = \langle q_1, l, q_2 \rangle \in E$.

Mit dieser Definition ergibt sich aus dem allgemeinsten Kommunikationspartner des Getränkeautomaten die Struktur des Kunde aus Abbildung Abbildung 6a. Die Definition definiert zusätzlich jede Markierung eines einzelnen Platzes als Endmarkierung.

Ein weitere Möglichkeit ist, Regionentheorie für Petrinetz [23] zu nutzen. Mit dieser erzeugen wir aus einem Transitionssystem ein Petrinetz, das Nebenläufigkeit zeigen kann. Da dieser Ansatz nicht direkt für offene Netze gedacht ist, müssen wir hier abschließend die Transitionen des Petrinetzes noch mit Kommunikationslabeln versehen. Dies ist jedoch problemlos möglich.

Mit diesen Überlegungen können wir Transitionssysteme und offene Netze analog betrachten. Insbesondere impliziert die Existenz eines allgemeinsten Kommunikationspartners die Existenz eines Controllers in Form eines offenen Netzes.

2.5 ZUSAMMENFASSUNG

Wir haben in diesem Kapitel alle grundlegende Begriffe eingeführt, die wir in allen folgenden Kapiteln benötigen.

Zentrale Begriffe sind *offene Netze* und deren *Komposition*, die wir aufgrund des Fokus dieser Arbeit in allen folgenden Kapiteln benötigen.

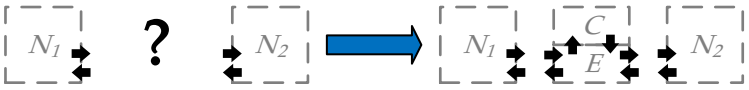
Wir haben weiterhin definiert, wie wir für ein gegebenes offenes Netz eine *universelle Umgebung* bilden und den *allgemeinsten Kommunikationspartner* ausrechnen. In den folgenden Kapiteln nutzen wir in der Regel lediglich aus, dass wir den allgemeinsten Kommunikationspartner ausrechnen können, aber nicht wie. In Kapitel 7 benutzen wir die Definition zum τ-Abschluss.

Teil I

DIE TECHNIK

3 ADAPTERSYNTHESE

Wie können wir zwei offene Netze adaptieren?
Wie trennen wir dabei Aspekte des
Datenflusses und des Kontrollflusses?



3.1 PROBLEMSTELLUNG

In diesem Kapitel führen wir unsere Technik zur Synthese eines Adapters offener Netze ein [39]. Zentraler Punkt dieser Technik ist die Trennung des Datenflusses vom Kontrollfluss.

Wie in der Einleitung bereits beschrieben, gibt es gerade bei einem Adapter verschiedene Aspekte, die für die Korrektheit von Bedeutung sind. Neben korrektem *Verhalten*, wie es zum Beispiel durch Controllersynthese im vorhergehenden Kapitel sichergestellt wird, spielt vor allem die *semantische Funktionalität* eines Adapters eine wichtige Rolle.

Der wohl wichtigste Anwendungsfall für ein offenes System ist der Austausch von Nachrichten. Eine Nachricht hat eine Bedeutung, die für das korrekte Funktionieren des Systems nicht beliebig geändert werden darf.

Wir wollen mit der hier vorgestellten Technik die beiden Punkte, semantische Funktionalität eines Adapters und dessen korrektes Verhalten, getrennt voneinander betrachten. Die semantische Funktionalität beschreiben wir mit einer semantischen Spezifikation bestehend aus *Transformationsregeln*. Eine Transformationsregel gibt an, wie Nachrichten manipuliert werden dürfen. Anschließend greifen wir auf die Technik zur Controllersynthese aus dem vorherigen Kapitel zurück, um sicherzustellen, dass ein Adapter ein gewünschtes Korrektheitskriterium erfüllt, indem er zielgerichtet die Transformationsregeln anwendet.

AUSGANGSSITUATION Gegeben seien $k \in \mathbb{N}$ offene Netze $N_1, \ldots, N_k$, eine semantische Spezifikation $\mathcal{R}$ der möglichen Nachrichtenmanipulationen in Form von Transformationsregeln und ein Korrektheitskriterium für offene Netze. Ziel ist es, einen Adapter zu generieren, dessen Datenfluss der semantischen Spezifikation genügt und dessen Kontrollfluss das Korrektheitskriterium sicherstellt.

GLIEDERUNG Wir führen zuerst Transformationsregeln als Baustein der semantischen Spezifikation in Abschnitt 3.2 ein. Mit Hilfe dieser Regeln generieren wir den ersten Teil des Adapters, die Engine, in Abschnitt 3.3 und stellen somit semantisch valides Verhalten des Adapters sicher. Bevor wir in 3.5 den Adapter mit einem Controller vervollständigen, der die Korrektheit des Verhaltens sicherstellt, diskutieren wir in Abschnitt 3.4 die Beschränkung der Engine, um die Controllersynthese überhaupt anwenden zu können. Wir schließen dieses Kapitel in Abschnitt 3.6 ab.

3.2 TRANSFORMATIONSREGEL

Wir haben in der Einleitung gesehen, dass das Modellieren des Datenflusses ein wichtiger Teil für die Komposition interagierender Systeme ist. Mit den Enterprise Integration Patterns [44] haben wir eine umfangreiche Sammlung zur Hand, um typische Anforderung an den Datenfluss auszudrücken.

Diese Patterns bieten einen großen Umfang an Konzepten zur Nachrichtenmanipulation und zum Leiten von Nachrichten zwischen Systemen. Diese Konzepte sind jedoch abstrakt und wir können sie auf einfachere Konzepte zurück führen. So lassen sich die Patterns zum Beispiel in gefärbte Petrinetze übersetzen [31], sodass uns in der Modellwelt ausreicht, dass wir Nachrichten konsumieren und produzieren können. Trotzdem können wir komplexe Konzepte des Datenflusses umsetzen.

In verschiedenen Arbeiten [10, 13, 15, 16, 28, 78] werden grundlegenden Anforderungen an die Nachrichtenmanipulation gestellt, um Datenfluss sinnvoll modellieren zu können. Zu diesen Anforderung gehört:

ERZEUGEN einer Nachricht: Es soll zum Beispiel für Protokollnachrichten wie einer Bestätigung möglich sein, oder wenn wir die Nachricht mit einer Standardwert belegen können. Für Nachrichten, die zum Beispiel persönliche Informationen enthalten, ist dies ausgeschlossen.

Kopieren einer Nachricht: Dies ist dann von Nöten, wenn ein beteiligtes System eine Information wiederholt benötigt, diese aber ursprünglich nur einmal zur Verfügung steht. Für sensible Daten kann dies auch ausgeschlossen sein.

Löschen einer Nachricht: Für eine elektronische Nachricht ist das fast immer möglich. Nur wenn sie rechtlich relevante Informationen, wie zum Beispiel bei einer Rechnung, oder ähnliches enthält, ist dies ausgeschlossen.

Transformieren einer Nachricht: Hauptanwendungsfall ist das Transformieren in einen anderen Typ, zum Beispiel, wenn sich zwei Systeme in ihren Schnittstellen unterscheiden. Die Transformation muss dabei bekannt sei. Dies kann sowohl die Struktur als auch den Inhalt einer Nachricht betreffen.

Aufteilen einer Nachricht: Dies erlaubt das Umwandeln komplexer Typen in ihre Bestandteile.

Zusammenfassen von Nachrichten: Dies ist die Umkehrung des Aufteilens und erlaubt die Erstellung komplexer Nachrichten.

Der zentrale Bestandteil eines Adapters ist dessen *semantische Spezifikation*. Diese soll sicher stellen, dass der Austausch von Nachrichten zwischen den beteiligten offenen Systemen die Bedeutung der Nachrichten respektiert. Dies drücken wir mit Hilfe von *Transformationsregeln* aus. Eine semantische Spezifikation besteht also aus Transformationsregeln.

Eine Transformationsregel beschreibt, wie Nachrichten eines Typs in Nachrichten eines anderen Typs umgewandelt werden. Wir wollen hier ganz explizit von der tatsächlichen Transformation abstrahieren, da wir in offenen Netzen vom konkreten Wert einer Nachricht abstrahieren. Die konkrete Transformation ist somit irrelevant. Wir können sie jedoch zusätzlich angeben, um eine spätere Implementation des Adapters, wie in Kapitel 5 beschrieben, zu ermöglichen.

Definition 23 (Transformationsregel)
Als *Transformationsregel* definieren wir folgenden Ausdruck:

$$R\colon x_1, \ldots, x_n \rightarrow y_1, \ldots, y_m,\ n, m \in \mathbb{N}$$

Tabelle 1: Ausdrucksmöglichkeiten einer Transformationsregel

Regelformat	Bedeutung
R_1: a → b	Umwandeln von Typ *a* in Typ *b*
R_2: → a	Erzeugen von Typ *a*
R_3: a → a, a	Kopieren von Typ *a*
R_4: a →	Löschen von Typ *a*
R_5: a, b, c → d, e	Transformieren von Typen *a*, *b*, *c* in Typen *d*, *e*
R_6: a → b, c, d	Aufteilen des Typs *a* in Typen *b*, *c*, *d*
R_7: a, b, c → d	Zusammenfassen der Typen *a*, *b*, *c* in Typ *d*

Dabei gibt R den Namen der Regel an, $x_1, \ldots, x_n$ sind die Nachrichtentypen, die beim Anwenden der Regel *benötigt* werden, und $y_1, \ldots, y_m$ sind die Nachrichtentypen, die beim Anwenden der Regel *erzeugt* werden.

Wir lassen zu, dass die Zahl der benötigten oder der erzeugten Nachrichten null ist.

NOTATION: Wir schreiben $c \in R$, wenn *c* ein Nachrichtentyp ist, der von der Transformationsregel R benötigt oder erzeugt wird.

Ein Nachrichtentyp darf auch mehrfach benötigt oder erzeugt werden. Dies bedeutet, dass wir eine entsprechende Anzahl verschiedener Nachrichten des gleichen Typs benötigen oder erzeugen. Die Liste der Typen darf zudem auch leer sein. Einige Beispiele für Transformationsregeln sehen wir in Tabelle 1. Diese spiegeln die oben genannten Anforderungen zur Manipulationen von Nachrichten wider.

Regeln von der Art R_1 stellen den Standardfall dar. Eine Nachricht muss übersetzt werden, damit sie einem anderen offenen System im richtigen Typ vorliegt. Dies könnte zum Beispiel die Umwandlung einer Temperatur von Grad Celsius in Grad Fahrenheit sein, oder aber auch die Umwandlung eines als Zeichenkette repräsentierten Wertes in eine Zahl.

Die Regeln der Art R_2 bis R_4 zeigen, dass wir Nachrichten eines bestimmten Typs auch erzeugen, kopieren oder löschen können. Dies betrifft in der Regel Protokollnachrichtentypen, die keinen Inhalt tragen, der von einem der beteiligten Systeme stammt. Typisch ist solch ein Fall bei einer Empfangsbestätigung oder bei einer Aus-

wahl. Andererseits soll zum Beispiel ein Nutzerkennwort nicht einfach erzeugt werden können. Eine Information, die explizit für einen zweites System bestimmt ist, darf nicht gelöscht werden, sondern muss dem anderen System zugestellt werden. Die Transaktionsnummer einer Bank sollte auch nicht einfach kopiert werden, schließlich ist sie zum einmaligen Gebrauch gedacht.

Das Beispiel von Regel R_5 zeigt, dass die Beziehung zwischen den Nachrichtentypen beliebig sein kann. Entsprechend zeigt Regel R_6, dass ein komplexer Nachrichtentyp in seine Teile zerlegt werden kann. Regel R_7 zeigt, dass einzelne Typen zu einem komplexeren Typ zusammengefasst werden können.

Mit mehreren Transformationsregeln lassen sich komplexere Konzepte beschreiben. Das Collapse- oder Gather-Pattern [28] beschreibt, wie wir eine nicht festgelegte, aber beschränkte Anzahl an Nachrichten vom Typ a in Typ b zusammenfassen. Wir realisieren diese Pattern mit den folgenden beiden Regeln: C_1: a $\rightarrow$ b, C_2: a, b $\rightarrow$ b. Die Idee ist, eine unvollständige Nachricht vom Typ b durch Nachrichten vom Typ a aufzufüllen. Die Regel C_1 erlaubt uns, initial eine Nachricht vom Typ a in den Typ b umzuwandeln. Mit Hilfe der zweiten Regel C_2 fassen wir jede weitere Nachricht vom Typ a mit der vorhandenen Nachricht vom Typ b zu einer Nachricht vom Typ b zusammen.

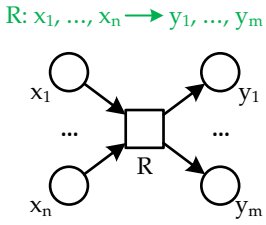


Abbildung 11

Wenn wir uns die Definition einer Transformationsregel (Definition 23) anschauen und sie mit einer Transition im Petrinetz vergleichen, stellen wir fest, dass sie konzeptionell eng verwandt sind. Dies bietet uns eine Möglichkeit zur Umsetzung einer Transformationsregel R: $x_1, \ldots, x_n \rightarrow y_1, \ldots, y_m$ wie in Abbildung 11 gezeigt. Die Transition R konsumiert Marken von den Plätzen x_1 bis x_n, wo die Regel R entsprechende Nachrichten vom Typ x_1 bis x_n benötigt. Analog produziert die Transition R Marken auf den Plätzen y_1 bis y_m, wo die Regel R Nachrichten vom Typ y_1 bis y_m erzeugt. Das Vorhandensein einer Marke auf einem Platz entspricht damit dem Vorhandensein einer Nachricht vom entsprechenden Typ, und das Schalten der Transition entspricht dem Effekt der Transformationsregel.

Damit sich ein Adapter semantisch korrekt verhält, d. h., die vorgenommenen Nachrichtentransformationen den Transformationsregeln entsprechen, müssen wir den Adapter im Wesentlichen so gestalten, dass er genau, wie oben gezeigt, die Transfor-

mationsregeln als Transitionen implementiert. Genau diese Funktion übernimmt die *Engine* eines Adapters.

3.3 ENGINE EINES ADAPTERS

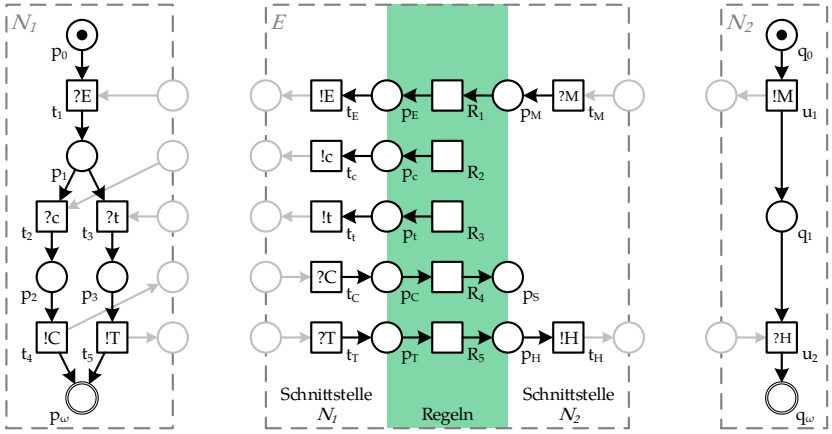
In der Engine eines Adapters findet die semantisch korrekte Verarbeitung von Nachrichten statt. Die Engine kommuniziert dazu mit den beteiligten offenen Netzen und implementiert die gegebenen Transformationsregeln. Wichtig ist hierbei, den Zusammenhang zwischen Kommunikationskanälen und Nachrichtentypen herzustellen.

Zur Definition der Engine führen wir zuerst deren Schnittstelle zu den gegebenen offenen Netzen ein. Wir setzen die Transformationsregeln als Transitionen um, die auf die mit den offenen Netzen ausgetauschten Nachrichten zugreifen. Da die Engine potentiell unbeschränktes Verhalten zeigen kann, betrachten wir abschließend noch, wie wir die Engine so beschränken, dass ein Controller die Einhaltung einer Schranke für die Plätze selbständig einhält.

In offenen Systemen werden oft Nachrichten unterschiedlichen Typs über den gleichen Kanal gesendet. Da wir von Daten abstrahieren, benutzen wir für jeden Nachrichtentyp einen eigenen Kanal, um die einzelnen Typen zu unterscheiden. Somit entspricht in unserem Modell ein Kommunikationskanal einem Nachrichtentyp.

Mit dieser Annahme verarbeitet eine Transformationsregel genau die Nachrichten, die über die entsprechenden Kommunikationskanal von einem der offenen Netze stammen, oder von einer anderen Transformationsregel erzeugt wurden. Jedoch müssen wir berücksichtigen, dass bei dieser Gleichsetzung von Nachrichtentyp und Kanal unser Kommunikationsmodell nicht verletzt wird. Die Transformationsregeln dürfen nicht auf den Kommunikationsplätzen arbeiten, die wir durch Komposition einführen.

Die Kommunikation zwischen zwei offenen Netzen ist nach Definition 11 gerichtet, d. h., ein offenes Netz kann entweder eine Marke auf einem Kommunikationsplatz produzieren oder konsumieren, aber nicht beides. Die Engine soll aber eine Nachricht, die sie selbst erzeugt hat, weiterverarbeiten können. Daher wollen wir zwischen der Kommunikation und der Transformation einer Nachricht unterscheiden, indem Transformationen ausschließlich auf internen Plätzen und nicht auf der bei der Komposition eingeführten Kommunikationsplätzen stattfinden.



(a) N_1 (b) E (c) N_2

Abbildung 12: Beispiel einer Engine E für die beiden offenen Netze N_1 und N_2 und Transformationsregeln $R_1, \ldots, R_5$

Für jeden Empfangskanal der Engine gibt es einen interne Kopie in Form eines Platzes. Auf diesem wird eine Marke nach dem Empfangen produziert, um die Anwesenheit einer Nachricht des entsprechenden Typs zu repräsentieren. Für jeden Sendekanal der Engine gibt es analog einen Platz, von dem eine Marke vor dem Senden Definition 11 wird. Die Nachrichtentypen einer Transformationsregel beziehen sich dann ausschließlich auf die internen Plätze der Engine.

Diese Idee erläutern wir an dem Beispiel in Abbildung 12. Dort sind zwei offene Netze N_1 und N_2 gegeben, die wir adaptieren wollen. Die Engine E stellt ein zu N_1 und N_2 passende Schnittstelle bereit und setzt die folgenden fünf Transformationsregeln um: R_1: M $\rightarrow$ E, R_2: $\rightarrow$ c, R_3: $\rightarrow$ t, R_4: C $\rightarrow$ S und R_5: T $\rightarrow$ H.

Die Netze und Regeln können wir als Getränkeautomaten mit Kunden interpretieren. Der Getränkeautomat N_1 in Abbildung 12a erwartet zuerst einen Euro E, danach wird extern eine Entscheidung getroffen. Empfängt der Automat ein c, dann soll der Automat eine Cola C ausgeben. Empfängt der Automat ein t, dann soll der Automat einen Tee T ausgeben. Der Kunde N_2 in Abbildung 12c stellt eine Münze M bereit und erwartet dann den Empfang eines Heißgetränks H.

Ein Adapter soll es uns ermöglichen, die unterschiedlichen Typen (zum Beispiel Münze und Euro) anzupassen. Außerdem soll er das richtige Getränk wählen, sodass der Kunde am Ende sein Heißgetränk erhält. Die Engine E in Abbildung 12b setzt dies um. Die Engine E stellt für jedes der beiden offenen Netze einen total komponierbaren Port zur Verfügung. So gibt es für jeden Kommunikationskanal in N_1 einen entsprechenden Kommunikationskanal in E (analog für N_2). Für jeden Kommunikationskanal gibt es zudem einen internen Platz, zum Beispiel für M den Platz p_M. Der Regel $R_1\colon M \to E$ entspricht die Transition R_1, die entsprechend vom Platz p_M konsumiert und auf den Platz p_E produziert.

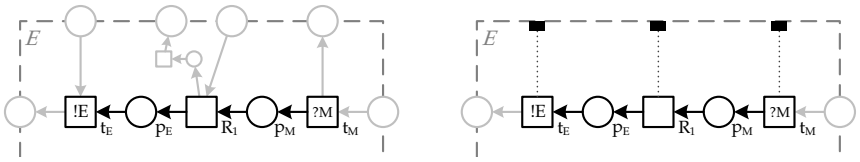
Mit den gegebenen Transformationsregeln und der daraus abgeleiteten Engine E können wir die beiden offenen Netze N_1 und N_2 adaptieren. Wenn der Kunde N_2 die Münze M sendet, können wir diese in der Engine mit t_M empfangen, Regel R_1 anwenden und einen Euro E an den Automaten N_1 senden. Wenden wir nun Regel R_3 an und senden die Nachricht t mit t_t an N_1, kann dieser einen Tee T senden. In der Engine empfangen wir diesen mit t_T, wandeln ihn mit Regel R_5 in ein Heißgetränk um und senden dieses mit t_H an N_2. Alle drei offenen Netze N_1, N_2 und E befinden sich dann in einer Endmarkierung.

Auch wenn wir mit der gezeigten Engine eine Endmarkierung erreichen können, gilt dies nicht für jede erreichbare Markierung. Die beiden Transitionen R_2 und R_3 haben beide jeweils einen leeren Vorbereich und können beliebig schalten. Demnach kann die Engine auch Cola als Getränk wählen, sodass der Kunde am Ende sein Heißgetränk nicht erhält. Daher müssen wir die Anwendung der Regeln sinnvoll steuern.

Ein Controller übernimmt die Aufgabe, die Transitionen der Engine so zu steuern, dass sie nur dann schalten, wenn ihr Schalten benötigt wird. Der Controller verhindert, dass durch wahlloses Schalten der Transitionen das gewünschte Korrektheitskriterium verletzt wird.

Damit ein Controller möglichst gut einschätzen kann, welche Transition er schalten kann, soll er nicht nur die Transitionen steuern, sondern auch über das Schalten der Transitionen informiert werden. Auf diese Weise kennt ein Controller den genauen Zustand einer Engine. Wir definieren zwei Möglichkeiten für die Kontrolle einer Engine: mit einer asynchronen Schnittstelle und einer synchronen Schnittstelle.

Bei einer *asynchronen Controllerschnittstelle* soll der Controller explizit das Senden einer Nachricht beziehungsweise das Anwenden einer Transformationsregel aktivieren können. Auch wenn eine Nachricht zum Senden bereit liegt, kann es sein, dass die



(a) Asynchron (b) Synchron

Abbildung 13: Schnittstelle einer Engine zum Controller, links asynchron und rechts synchron

Engine diese erst später tatsächlich senden darf, damit das entsprechende offene Netz keine Markierung erreicht, die das Korrektheitskriterium verletzt. Wenn es mehrere Alternativen gibt, Transformationsregeln anzuwenden, dann wählt, wie oben im Beispiel gesehen, der Controller nur solche, die nicht dazu führen, dass das Korrektheitskriterium verletzt wird. Über das Empfangen einer Nachricht und die Anwendung einer aktivierten Transformationsregel soll der Controller informiert werden. Damit erhält der Controller einen Überblick, welche Nachrichten in der Engine vorliegen, und welche Regeln er als nächstes anwenden beziehungsweise welche Nachrichten er senden kann.

Auch bei einer *synchronen Controllerschnittstelle* soll der Controller die Transitionen aktivieren und Informationen über das Schalten erhalten können. Da die synchrone Kommunikation jedoch ungerichtet ist, erhält jede Transition einen eigenen synchronen Kanal zum Controller.

Ein Beispiel für Controllerschnittstellen sehen wir in Abbildung 13. Dort sind für unser Automatenbeispiel der obere Teil der Engine mit einer entsprechenden Schnittstelle ausgestattet. Die restlichen Transitionen stellen wir uns analog vor.

In Abbildung 13a sehen wir eine asynchrone Controllerschnittstelle. Die Transition t_M verfügt über einen nach oben gerichteten Sendekanal zum Controller. Über diesen kann die Transition den Controller informieren, dass eine Nachricht M von der Engine empfangen wurde. Wird die Transition R_1 nun über den Empfangskanal vom Controller aktiviert, kann diese Schalten und informiert den Controller über den Sendekanal, dass sie geschaltet und M in E umgewandelt hat. Da wir in 9 eines offenen Netzes nur ein Kommunikationslabel pro Port zulassen, fügen wir noch einen Platz und Transition ein, die für das Senden zuständig ist. Zuletzt kann der Controller die Transition t_E

über einen Empfangskanal aktivieren, sodass die Nachricht vom Typ E von der Engine gesendet werden darf.

Die synchrone Controllerschnittstelle in Abbildung 13b verhält sich analog zum asynchronen. Da die synchrone Kommunikation hier jedoch ungerichtet ist, unterscheiden wir nicht zwischen Aktivieren und Informieren. Daher hat jede Transition genau einen synchronen Kanal.

Für eine gegebene Menge zu adaptierender offener Netze $N_1, \ldots, N_k$ ($k \in \mathbb{N}$) und eine Menge von Transformationsregeln $\mathcal{R}$ als semantische Spezifikation definieren wir nun die *Engine eines Adapters*. Wir setzen voraus, dass die offenen Netze paarweise disjunkt sind, d. h., weder Transitionen noch Plätze eines offenen Netzes den gleichen Namen haben wie Transitionen oder Plätze eines anderen offenen Netzes. Dies können wir durch Umbenennen von Transitionen und Plätzen sicher stellen.

Definition 24 (Engine eines Adapters)
Sei $N_1, \ldots, N_k$ eine Menge von k paarweise disjunkten offenen Netzen mit $N_i = \left\langle P_i, T_i, F_i, \alpha_i, \Omega_i, \{port_{N_i}\}, \lambda_i \right\rangle$, $i = 1, \ldots, k$, und $\mathcal{R}$ eine Menge an Transformationsregeln. Dann definieren wir als *Engine eines Adapters* das offene Netz $E = \langle P, T, F, \alpha, \Omega, \mathcal{I}, \lambda \rangle$ mit

$$P = \{p_c \mid \exists i \in \{1, \ldots, k\} : !c \text{ od. } ?c \in port_{N_i}\} \cup \{p_c \mid c \in \mathrm{R} \in \mathcal{R}\},$$

$$T = \{t_c \mid \exists i \in \{1, \ldots, k\} : !c \text{ od. } ?c \in port_{N_i}\} \cup \{t_\mathrm{R} \mid \mathrm{R} \in \mathcal{R}\},$$

$$\begin{aligned} F = \quad & \{(p_c, t_c) \mid \exists i \in \{1, \ldots, k\} : ?c \in port_{N_i}\} \\ & \cup \{(t_c, p_c) \mid \exists i \in \{1, \ldots, k\} : !c \in port_{N_i}\} \\ & \cup \{(p_c, t_\mathrm{R}) \mid \mathrm{R} \text{ benötigt } c\} \cup \{(t_\mathrm{R}, t_c) \mid \mathrm{R} \text{ erzeugt } c\}, \end{aligned}$$

$$\alpha = [],$$

$$\Omega = \{[]\},$$

$\mathcal{I} = \{port_1, \ldots, port_k, port_C\}$, wobei sich die Ports kanonisch aus folgender Labelfunktion ergeben,

λ ist für $port_1, \ldots, port_k$ definiert als

$$\lambda(port_i, t_c) = \begin{cases} !c & \text{falls } ?c \in port_{N_i} \\ ?c & \text{falls } !c \in port_{N_i} \\ \tau & \text{sonst.} \end{cases}$$

Die Labelfunktion für den Port $port_C$ eines Controllers ist abhängig davon, ob wir eine asynchrone oder synchrone Schnittstelle betrachten:

Asynchrone Schnittstelle

Damit eine Regeltransition sowohl aktiviert werden kann, als auch den Controller informieren kann, fügen wir je eine Transition t'_R und einen Platz p_R für alle $R \in \mathcal{R}$ in die Engine mit den zusätzlichen Kanten (t_R, p_R) und (p_R, t'_R) ein. Die Labelfunktion ist dann

$$\lambda(port_C, t) = \begin{cases} !inf_t & \text{falls } t = t_c, \exists i \in \{1, \ldots, k\} : !c \in port_{N_i} \\ ?akt_t & \text{falls } t = t_c, \exists i \in \{1, \ldots, k\} : ?c \in port_{N_i} \\ ?akt_t & \text{falls } t = t_R, R \in \mathcal{R} \\ !inf_t & \text{falls } t = t'_R, R \in \mathcal{R} \end{cases}$$

Synchrone Schnittstelle

$\lambda(port_C, t) = \#sync_t$

NOTATION Wir schreiben $E = Engine(N_1, N_2, \mathcal{R})$ um E als Engine gemäß der Definition festzulegen.

Für die asynchrone Controllerschnittstelle bedeutet $!inf_t$, dass ein Controller beim Schalten von t *informiert* wird, und über $?akt_t$ kann ein Controller t *aktivieren*. Entsprechend erhält eine Regeltransition das Label $?akt_t$ als auch die eine zusätzliche Transition mit Label $!inf_t$.

Die Unterschiede, die sich aus der Wahl einer asynchronen oder synchronen Schnittstelle zwischen Engine und einem Controller ergeben, betrachten wir im nächsten Kapitel.

3.4 BESCHRÄNKEN DES ZUSTANDSRAUMS

Bei der Berechnung eines Controllers gibt es ein Problem, welches die Beschränktheit eines offenen Netzes betrifft. Der vorgestellte Algorithmus zur Controllersynthese setzt ein endliches System als Eingabe voraus. Die Engine zeigt mit Definition 24 jedoch potentiell unbeschränktes Verhalten.

Eine Regel, die Nachrichten erzeugt, wie R_2: $\rightarrow$ c im Beispiel, hat keine weiteren Vorbedingungen und kann beliebig schalten. Der entsprechende Platz für den Nachrichtentyp kann somit unbeschränkt viele Marken anhäufen. Da die Algorithmen zur Controllersynthese von einem endlichen System ausgehen, terminieren in solchen Situationen nicht. Damit wir solche Algorithmen nutzen können, müssen wir eine sinnvolle Beschränkung der Plätze erzwingen.

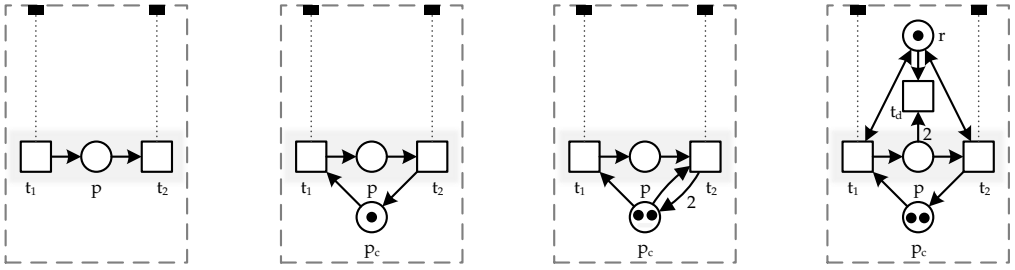
Die Beschränkung betrifft zwei Arten von Plätzen: Die Nachrichtentypplätze in der Engine, und die Kommunikationsplätze zwischen den gegebenen offenen Netzen und der Engine. Auf letzteren sollte die vorgegebene Kommunikationsschranke aus Definition 13 eingehalten werden.

Anhand eines Beispiels wollen wir zeigen, welche Möglichkeiten wir haben, die Anzahl an Marken auf einem Platz zu beschränken, indem wir die Verletzung dieser Schranke als Verletzung des gewählten Korrektheitskriterium modellieren.

Die offenen Netze in Abbildung 14 sollen uns dabei als minimales Beispiel dienen. Von links nach rechts sehen wir das unbeschränkte Ausgangsnetz mit synchroner Schnittstelle (Abbildung 14a), das Netz erweitert um einen Komplementärplatz (Abbildung 14b), das Netz erweitert für den Fall Schwache Terminierung (Abbildung 14c) und das Netz erweitert für den Fall Verklemmungsfreiheit (Abbildung 14d).

In Abbildung 14a haben wir ein Beispiel eines unbeschränkten Platzes p. Um eine Anzahl k an Marken auf p zu erzeugen, muss ein Partner sich k mal mit der Transition t_1 synchronisieren. Wir nehmen für das Beispiel an, dass p 1-beschränkt sein soll. Das bedeutet, dass t_1 nicht zweimal hintereinander schalten darf, ohne dass t_2 dazwischen schaltet.

Eine Möglichkeit, die Anzahl an Marken auf einem Platz p zu beschränken, ist die Einführung eines Komplementärplatzes p_c, wie in Abbildung 14b zu sehen. Für eine Schranke k an Marken soll die Anzahl an Marken auf p und p_c zusammen immer genau k betragen. Dafür muss jede Transition, die eine Marke auf p erzeugt, eine Marke von p_c konsumieren, und jede Transition, die von p eine Marke konsumiert, muss eine



(a) Controller gesteuert (b) Komplementärplatz (c) Schwache Terminierung (d) Verklemmungsfreiheit

Abbildung 14: Beschränken eines unbeschränkten Platzes

Marke auf p_c produzieren. Der Platz p_c gibt somit vor, wie viele Marken noch auf p erzeugt werden dürfen, bevor die Schranke erreicht ist.

Die Einführung von Komplementärplätzen ist jedoch nur für die Plätze der Engine realistisch, nicht jedoch für die Kommunikationsplätze zwischen Engine und den gegebenen offenen Netzen. Nach der Komposition könnten wir auch für einen Kommunikationsplatz einen Komplementärplatz einfügen, jedoch entspricht dies nicht mehr unserem Kommunikationsmodell. Wir müssen daher sicher stellen, dass ein Adapter mit Hilfe des Controllers die Kommunikationsschranke auch ohne Einführen von Komplementärplätzen einhält.

Für Schwache Terminierung und Verklemmungsfreiheit stellen wir je eine Variante vor, wie wir einen Controller dazu bringen, eine Schranke k selbständig einzuhalten. Für ein unbeschränktes offenes Netz N konstruieren wir ein beschränktes offenes Netz N', in dem die Verletzung der Schranke k zur Verletzung des Korrektheitskriteriums führt. Ein Controller ist höchstens dann korrekt, wenn er die Schranke k einhält. Wir zeigen, dass für die beiden genannten Korrektheitskriterien, dass ein so berechneter Controller auch in N das Einhalten der Schranke k sicher stellt.

3.4.1 *Schwache Terminierung*

Wie wir ein offenes Netz N beschränken können, sodass ein Controller letztendlich sicherstellt, dass eine Schranke auf einem Platz nie überschritten wird, können wir für

den Fall Schwache Terminierung in Abbildung 14c sehen. Wir arbeiten auch hier mit einem Komplementärplatz p_c, allerdings erlaubt dieser, die Schranke auf dem Platz p zu verletzen. Eine Verletzung bedeutet, dass das Netz ab der entsprechenden Markierung nicht mehr schwach terminierend ist.

Wenn wir eine Schranke k (im Beispiel $k = 1$) für p annehmen, dann legen wir eine Marke mehr auf den Komplementärplatz p_c. Die Transition t_1 hat dann die Möglichkeit, die Schranke für p zu verletzen, indem sie $k + 1$-mal schaltet. Die Transition t_2 legt wie bei einem Komplementärplatz üblich für jede Marke, die sie von p konsumiert, eine Marke auf p_c zurück. Gleichzeitig möchten wir, dass t_2 nur dann schaltet, wenn die Schranke noch nicht verletzt wurde. Die Schranke ist nicht verletzt, wenn wenigstens eine Marke auf p_c liegt. Daher konsumiert t_2 eine Marke von p_c und legt sie sofort zurück, sodass sich die Kantenvielfachheit 2 für die Kante (t_2, p_c) ergibt. Wenn die Verletzung der Schranke eintritt, kann t_2 nicht mehr schalten und somit die Verletzung nicht rückgängig machen.

Sobald die Schranke k für p verletzt ist, können wir weder t_1 noch t_2 schalten, weil keine Marke mehr auf p_c liegt. Da die Endmarkierungen sinnvoller weise maximal k Marken auf p voraussetzen, und die $k + 1$ Marken auf p nicht mehr entfernt werden, bedeutet das Verletzen der Schranke, dass kein Endzustand erreichbar ist. Das nach der obigen Anleitung konstruierte offene Netz ist endlich, weil maximal $k + 1$ Marken auf jedem einzelnen Platz liegen können.

Lemma 25 (Beschränkung für Schwache Terminierung)
Sei N ein offenes Netz, N' das mit der oben genannte konstruierten Beschränkung abgeleitete offene Netz für eine Schranke k und C ein Controller für N', sodass $N' \oplus C$ schwach terminierend ist. Dann ist keine Markierung in $N' \oplus C$ erreichbar, in der auf einem Platz mehr als k Marken liegen.

Beweis.
Angenommen, es gäbe eine von der Anfangsmarkierung erreichbare Markierung μ in $N' \oplus C$, sodass für einen Platz p in N gilt, dass $\mu(p) > k$. Mit der angegebenen Konstruktion bedeutet das aber, dass $k + 1$ Marken auf p liegen und somit keine Marken auf dessen Komplementärplatz p_c. Somit können weder die Vor- noch die Nachtransitionen schalten und p behält seine $k + 1$ Marken, was in keiner Endmar-

kierung erlaubt ist. Das steht im Widerspruch zur Voraussetzung der schwachen Terminierung. ∎

Die angegebene Konstruktion lässt auch zu, dass wir für jeden Platz eine individuelle Schranke vorgeben. Die Gültigkeit des Lemmas ändert sich dadurch nicht, da die Konstruktion lokal für einen Platz und dessen Vor- und Nachbereich ist. Insbesondere können wir die Konstruktion nutzen, um die Kommunikationsschranke zwischen offenen Netzen zu erzwingen. Wir benötigen die Konstruktion nur für die Controllersynthese. Wenn der Controller existiert, dann können wir ihn mit dem ursprünglichen offenen Netz und somit den definierten Kommunikationsmodell ohne diese Konstruktion nutzen, wie folgendes Korollar zeigt.

Korollar 26
Wenn $N' \oplus C$ schwach terminierend ist, dann ist auch $N \oplus C$ schwach terminierend und in keiner erreichbaren Markierung liegen mehr als k Marken auf einem Platz von N.

Beweis.
Das offene Netz N' unterscheidet sich von N nur dadurch, dass wir für jeden Platz einen Komplementärplatz mit entsprechenden Kanten eingefügt haben. Damit schränken wir das Verhalten von N' gegenüber N höchstens ein. Somit ist jedes Verhalten, das $N' \oplus C$ zeigen kann, auch in $N \oplus C$ möglich. Außerdem wissen wir per Konstruktion von N', dass in keiner erreichbaren Markierung von $N' \oplus C$ mehr als k Marken auf einem Platz liegen können. Die Markierungen von $N \oplus C$ unterscheiden sich nur durch die fehlenden Komplementärplätze, und somit liegen in keiner erreichbaren Markierung von $N \oplus C$ mehr als k Marken auf einem Platz. ∎

3.4.2 *Verklemmungsfreiheit*

Eine Verklemmung ist eine Markierung, die weder eine Nachfolgemarkierung besitzt noch selbst eine Endmarkierung ist. *Verklemmungsfreiheit* eines offenen Netzes bedeutet, dass von der Anfangsmarkierung keine Verklemmung erreichbar ist.

Wenn wir als Korrektheitskriterium Verklemmungsfreiheit fordern, dann können wir ähnlich vorgehen wie für Schwache Terminierung. Nur dass es im Falle einer Verletzung nicht ausreicht, dass eine Endmarkierung nicht mehr erreicht werden kann, sondern es muss genau dann eine Verklemmung erreichbar sein. Das Beispiel für die Umsetzung sehen wir in Abbildung 14d.

Auch für Verklemmungsfreiheit wollen wir einen Platz p grundsätzlich über einen Komplementärplatz p_c beschränken, aber die Verletzung der Schranke k (im Beispiel $k = 1$) zulassen. Insofern enthält p_c $k + 1$ Marken. Die Transition t_1 kann die Schranke auf p verletzen, indem sie $k + 1$ mal schaltet, ohne dass t_2 schaltet. Im Falle einer Verletzung ist die Transition t_d aktiviert, die $k + 1$ Marken von p konsumiert und die Marke von r entfernt.

Für jede Transition aus N haben wir eine Schlaufe zum Platz r eingefügt, das heißt, jede Transition kann nur schalten, wenn auf r ein Marke liegt, welche die Transition auch wieder produziert. Wenn die Verklemmungstransition t_d schaltet, dann wird die Marke von r entfernt, und keine Transition ist mehr aktiviert – das Netz ist in einer Verklemmung.

Allein die Möglichkeit, dass mit $k + 1$ auf p eine Verklemmung möglich ist, ist ausreichend, auch wenn wir das Schalten von t_d nicht erzwingen können. Theoretisch können anderen Transitionen beliebig weiter schalten, ohne dass wir tatsächlich eine Verklemmung erreichen. Aber allein die Möglichkeit einer Verklemmung widerspricht bereits der Verklemmungsfreiheit.

Lemma 27 (Beschränkung für Verklemmungsfreiheit)
Sei N ein offenes Netz, N' das mit der oben genannte konstruierten Beschränkung abgeleitete offene Netz für eine Schranke k und C ein Controller für N', sodass $N' \oplus C$ verklemmungsfrei ist. Dann ist keine Markierung in $N' \oplus C$ erreichbar, in der auf einem Platz mehr als k Marken liegen.

Beweis.
Angenommen, es gäbe eine von der Anfangsmarkierung erreichbare Markierung μ in $N' \oplus C$, sodass für einen Platz p in N gilt, dass $\mu(p) > k$. Mit der angegebenen Konstruktion bedeutet das aber, dass $k + 1$ Marken auf p liegen und somit die Transition t_d erstmalig schalten kann. Das Schalten von t_d entfernt jedoch die Marke vom

Platz r und es kann keine weitere Transition schalten. Da wir den Platz r einfügen, legen wir fest, dass in jeder Endmarkierung des ursprünglichen Netzes der Platz r markiert sein muss. Nach dem Schalten von t_d haben wir eine Markierung erreicht, in der keine Transition schalten kann, und die keine Endmarkierung ist. Somit ist diese Markierung eine Verklemmung im Widerspruch zur Voraussetzung. ∎

Auch diese Konstruktion lässt zu, dass wir für jeden Platz eine individuelle Schranke vorgeben.

Korollar 28
Wenn $N' \oplus C$ verklemmungsfrei ist, dann ist auch $N \oplus C$ verklemmungsfrei und in keiner erreichbaren Markierung liegen mehr als k Marken auf einem Platz von N.

Beweis.
Das offene Netz N' unterscheidet sich von N nur dadurch, dass wir für jeden Platz einen Komplementärplatz mit entsprechenden Kanten, eine Transition zum Erreichen einer Verklemmung und einen Platz r eingefügt haben. Damit schränken wir das Verhalten von N' gegenüber N höchstens ein. Somit ist jedes Verhalten, das $N' \oplus C$ zeigen kann, auch in $N \oplus C$ möglich. Außerdem wissen wir per Konstruktion von N', dass in keiner erreichbaren Markierung von $N' \oplus C$ mehr als k Marken auf einem Platz liegen können. Die Markierung von $N \oplus C$ unterscheiden sich nur durch die fehlenden Komplementärplätze und r, und somit liegen in keiner erreichbaren Markierung von $N \oplus C$ mehr als k Marken auf einem Platz. ∎

Die zweite Konstruktion funktioniert auch für Schwache Terminierung. Allerdings wird das offene Netz dadurch wesentlich komplexer und wir verlieren die Nebenläufigkeit, das unabhängige Schalten der Transitionen, da alle Transitionen über den Platz r in Konflikt stehen.

Welche Art von Beschränkung wir bezüglich eines bestimmten Korrektheitskriterium wählen, hängt nicht nur von dem Kriterium selbst, sondern auch vom verwendeten Synthesealgorithmus beziehungsweise dessen Implementierung ab, die unterschiedlich auf die Netzanpassungen reagieren könnte.

3.5 CONTROLLERSYNTHESE

Für eine Menge offener Netze $N_1, \ldots, N_k$ und eine Menge $\mathcal{R}$ von Transformationsregeln können wir bereits eine Engine E definieren. Diese Engine stellt sicher, dass nur semantisch valides Verhalten in der Interaktion zwischen den offenen Netzen auftreten kann. Mit der Schnittstelle zu einem Controller in E haben wir zudem die Möglichkeit geschaffen, das Verhalten der Engine soweit einzuschränken, dass ein gegebenes Korrektheitskriterium erfüllt wird. Durch Beschränkung der Engine stellen wir sicher, dass wir Algorithmen zur Controllersynthese benutzen können, die nur auf endlichen Systemen funktionieren.

Durch einen Controller ergänzen wir eine Engine zu einem Adapter.

Definition 29 (Adapter)
Seien $N_1, \ldots, N_k$ eine Menge offener Netze und $\mathcal{R}$ eine Menge von Transformationsregeln. Sei E eine Engine gemäß Definition 24. Für ein gegebenes Korrektheitskriterium sei C ein Kommunikationspartner von $(\ldots(E.port_1 \oplus N_1.port_{N_1}).port_2 \oplus \ldots).port_k \oplus N_k.port_{N_k}$, sodass die Komposition mit C das Korrektheitskriterium erfüllt. Dann nennen wir $E \oplus C$ einen *Adapter* für $N_1, \ldots, N_k$.

Mit Hilfe von Definition 19 können wir ein solches C sogar ausrechnen, falls eines existiert. Für eine gegebene Menge offener Netze und Transformationsregeln können wir somit entscheiden, ob die offenen Netze bezüglich der Transformationsregeln bezüglich Schwacher Terminierung adaptieren können oder nicht. Für den Fall, dass kein Controller existiert, betrachten wir in 7 Möglichkeiten, ob es Möglichkeiten gibt, durch Hinzufügen von Transformationsregeln doch einen Controller und somit Adapter synthetisieren zu können.

In Abbildung 15 sehen wir einen Controller für das Beispiel des Getränkeautomaten. Als synchrone Label benutzen wir die Namen der Transitionen der Engine, d. h., jede Transition in C synchronisiert sich mit der entsprechenden Transition in E.

An der Struktur des Controllers sehen wir, dass er genau den korrekten Ablauf der Engine wiedergibt, den wir vorhin bereits beschrieben haben. Die Münze vom Kunden wird empfangen (t_M), mit Regel R_1 in einen Euro umgewandelt und an den Getränkeautomaten gesendet (t_E). Die Regel R_3 wählt den Tee aus und sendet diese Wahl an den Getränkeautomaten (t_t). Anschließend wird der Tee empfangen (t_T), mit

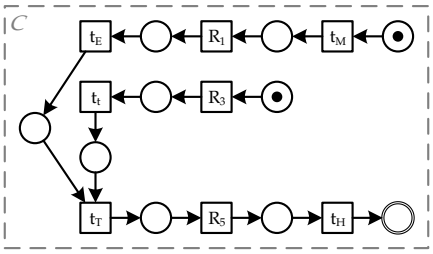


Abbildung 15: Controller für die Engine des Getränkeautomatenbeispiels, die Transitionen synchronisieren sich mit den gleichnamigen Transitionen der Engine

Regel R_5 in ein Heißgetränk umgewandelt und an den Kunden gesendet (t_H). Damit befindet sich der Controller in einer Endmarkierung.

Insbesondere nutzt der Controller jede dieser Transitionen genau ein mal und die Transitionen, die zur Wahl von Cola führen, nie. Dies ist also genau das Verhalten, das wir uns gewünscht haben, d. h., Kunde und Getränkeautomat erreichen eine Endmarkierung.

3.6 Zusammenfassung

In diesem Kapitel haben wir die zentrale Technik zur *Adaptersynthese* eingeführt. Sie beruht auf der strikten Trennung von Daten- und Kontrollfluss.

Validen Datenfluss spezifizieren wir mit Hilfe einer semantischen Spezifikation, die einer Menge von *Transformationsregeln* entspricht. Mit Hilfe der Transformationsregeln definieren wir die *Engine E* eines Adapters, die einerseits die Schnittstelle des Adapters zu den gegebenen offenen Netzen enthält, und andererseits die Transformationsregeln implementiert.

Anschließend haben wir Möglichkeiten betrachtet, das Verhalten einer Engine zu beschränken, um Algorithmen zur Controllersynthese anwenden zu können.

Wir nutzen die Technik zur Controllersynthese aus Kapitel 2 um einen *Controller C* zu berechnen und somit den *Adapter* $E \oplus C$ zu vervollständigen.

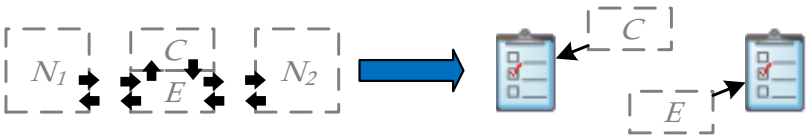
Die Implementation dieser Technik erfolgt im Werkzeug Marlene (siehe 5).

Im folgenden Kapitel betrachten wir, welche Eigenschaften die vorgestellte Technik zu Adaptersynthese hat, und wie wir die Trennung von Daten- und Kontrollfluss

ausnutzen können. In Kapitel 6 zeigen wir, wie wir einen synthetisierten Adapter verteilen.

4 EIGENSCHAFTEN DER TECHNIK

Wann existiert ein Adapter? Welche Eigenschaften besitzt die Technik, Kontroll- und Datenfluss zu trennen, und wie können wir sie ausnutzen?



4.1 PROBLEMSTELLUNG

Wir betrachten in diesem Kapitel, welche speziellen Eigenschaften die in Kapitel 3 vorgestellte Technik hat. Dazu wollen wir insbesondere Möglichkeiten der Engine und des Controllers eines Adapters ausloten.

Zuerst gehen wir der Frage nach, wann es möglich ist, für zwei gegebene offene Netze einen Adapter zu generieren. Im vorherigen Kapitel haben wir eine Technik zur Controllersynthese für offene Netze eingeführt. Diese stellt jedoch nicht die Existenz eines Controllers für ein offenes Netz sicher.

Wir geben zwei notwendige Voraussetzungen für die Existenz eines Controllers und somit Adapters an. Zum einen müssen die gegebenen offenen Netze selbst jeweils einen Controller besitzen, da anderenfalls kein Adapter existiert. Zum anderen hängt die Existenz eines Adapters von der semantischen Spezifikation ab. Wir zeigen, dass es immer eine Menge an trivialen Transformationsregeln gibt, um einen Adapter zu generieren. Allerdings lassen wir bei dieser Betrachtung die Semantik der Regeln außer Acht. Welche Möglichkeiten wir haben, Rückschlüsse aus der Nichtexistenz eines Controllers auf fehlende Transformationsregeln zu ziehen, betrachten wir in Kapitel 7.

In diesem Zusammenhang betrachten wir auch, inwieweit die Wahl der Schnittstelle zwischen Engine und Controller eines Adapters Auswirkungen auf die Existenz eines Adapters hat. In Definition 24 der Engine definieren wir sowohl eine synchrone und asynchrone Schnittstelle.

Anschließend erweitern wir die Möglichkeiten der Technik. Wir haben bereits im vorherigen Kapitel gesehen, dass wir durch Änderungen an der Engine eine Eigenschaft des Adapters, wie die Beschränkung der Plätze, beschreiben können. Wir betrachten in diesem Kapitel das Beispiel eines Firewallsystems, um die konzeptuelle Flexibilität des vorgestellten Ansatzes zu zeigen. Eine Firewall hat die Aufgabe, Nachrichten zwischen einem Server und einem Client zu filtern. Gute Nachrichten sollen die Firewall passieren, schlechte Nachrichten müssen verworfen werden. Wir modellieren die Filtermöglichkeiten als Datenfluss eines Adapters, passen die Kommunikationssemantik der einer Firewall an, und berechnen einen Controller und die Filterung zu steuern.

Mit der im letzten Kapitel vorgestellten Technik zur Adaptersynthese wird ein Adapter synthetisiert, der möglichst viel Verhalten zeigt. Obwohl das Verhalten per Konstruktion korrekt ist, ist nicht jeder Aspekt des Verhaltens unbedingt erwünscht. Da wir die gegebenen offenen Netze nicht verändern wollen, müssen wir einen Weg finden, um zum Beispiel Verhalten ausschließen oder zu erzwingen. Wenn wir einen Onlineshop adaptieren, könnte eine Anforderung sein, dass ein bestimmtes Zahlungsmittel genutzt oder eine Versandart ausgeschlossen wird. Für die uns benutzt Controllersynthese gibt es Erweiterungen, die uns genau dies erlauben.

Falls im Voraus jedoch nicht klar ist, ob wir bestimmtes Verhalten ausschließen können, ohne die Existenz eines Adapters zu gefährden, entwickeln wir ein Optimierungsproblem, das den Adapter nach der Synthese noch einschränken kann. Der Wunsch eines Onlineshops ist es sicher, dass ein Kunde ein Bestellvorgang komplett zu Ende führt. Gleichzeitig bietet der Onlineshop die Möglichkeit einen Bestellvorgang abzubrechen. Die letztere Möglichkeit zu verbieten erlaubt keine formal korrekte Interaktion mit Kunden, denen zum Beispiel ein benötigtes Zahlungsmittel fehlt. Über die Optimierung des synthetisierten Adapters schließen wir unerwünschtes Verhalten aus, wenn es nicht für die korrekte Interaktion notwendig ist.

AUSGANGSSITUATION Gegeben sei eine nicht-leere Menge offener Netze. Weiterhin sei eine semantische Spezifikation mit Transformationsregeln gegeben, wobei wir teil-

weise rein technisch motivierte Transformationsregeln ohne semantische Bedeutung benutzen.

GLIEDERUNG Der Abschnitt 4.2 betrachtet die Frage, in welchen Fällen die im vorherigen Abschnitt vorgestellte Technik zu Adaptersynthese einen Adapter als Ergebnis liefert, beziehungsweise warum dies nicht möglich ist. Im Abschnitt 4.3 untersuchen wir die Möglichkeit, die Technik auf andere Szenarien anzuwenden und zeigen, wie durch flexible Anpassung der Engine dem Szenario entsprechende Ergebnisse erzielt werden können. Zuletzt nutzen wir in Abschnitt 4.4 Resultate, welche die Controllersynthese beeinflussen, und wenden diese auf Adapter an. Die Ergebnisse fassen wir in Abschnitt 4.5 zusammen.

4.2 ALLGEMEINE EIGENSCHAFTEN

Wir haben im vorherigen Kapitel für zwei offene Netze N_1 und N_2 und eine Menge $\mathcal{R}$ von Transformationsregeln eingeführt, wie wir einen Adapter synthetisieren. Aus der Eingabe erzeugen wir eine Engine $E = \mathit{Engine}(N_1, N_2, \mathcal{R})$ und für $N_1 \oplus E \oplus N_2$ einen Controller C.

Die Existenz eines Controllers und somit eines Adapters ist dabei aber nicht sicher gestellt. Wir betrachten, welche Eigenschaften die offenen Netze N_1 und N_2 als auch die Regelmenge $\mathcal{R}$ erfüllen müssen, damit ein Adapter generiert werden kann. Wir zeigen zum einen, dass jedes der beiden der Netze selbst kontrollierbar sein muss, anderenfalls existiert kein Adapter. Zum anderen stellen wir fest, dass wir für zwei kontrollierbare offene Netze immer einen Adapter generieren können, falls wir eine entsprechende Menge $\mathcal{R}$ an Transformationsregeln gegeben haben.

Während die Eingabe ganz offensichtlich Einfluss auf die Existenz eines Adapters hat, haben wir in Definition 24 zudem noch die Möglichkeit, die Form der Engine zu beeinflussen, indem wir entweder eine synchrone oder eine asynchrone Schnittstelle zwischen Engine und Controller wählen. Mooij und Voorhoeve [74] zeigen, dass konzeptuell beide Schnittstellenarten für die gleichen Eingaben zum Adapter führen. Das Ergebnis gilt für potentiell unbeschränkte Netze. Die Controllersythese, die wir benutzen, arbeitet aber ausschließlich auf beschränkten Netzen. Die Autoren führen fort, dass in diesem Fall die Wahl der synchronen Schnittstelle häufiger zu einem Adapter führt.

Für die Art der Beschränkung, wie wir sie im vorherigen Kapitel eingeführt haben, stellen wir jedoch fest, dass für eine geeignete Schranke die beiden Schnittstellenarten wieder für die gleichen Eingaben zu einem Adapter führen.

4.2.1 *Existenz eines Adapters*

Eine wichtige Voraussetzung für die Anwendbarkeit der vorgestellten Technik ist, dass für die beteiligten offenen Netze selbst jeweils ein Controller existiert. Ist dies für eines der offenen Netze nicht der Fall, existiert kein Adapter.

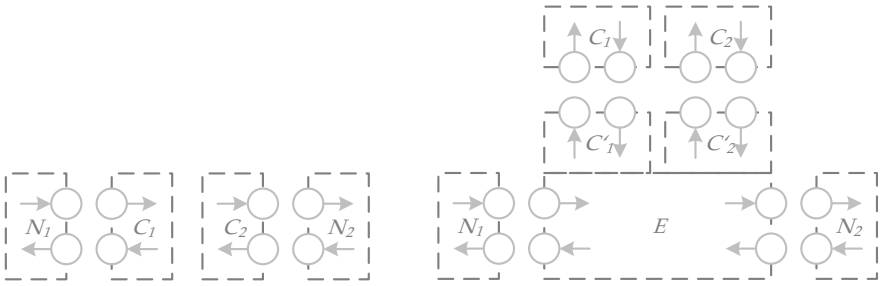
Lemma 30 (Nicht-Kontrollierbarkeit und Existenz eines Adapters)
Wenn für ein offenes Netz N_1 kein Controller existiert, dann existiert kein Adapter für N_1.

Beweis.
Seien N_1 und N_2 zwei offene Netze, $\mathcal{R}$ eine Menge von Transformationsregeln und $E = \mathit{Engine}(N_1, N_2, \mathcal{R})$ die entsprechende Engine. Wir nehmen ein beliebiges Korrektheitskriterium an, und dass für N_1 kein Controller existiert.

Angenommen, wir finden für N_1 und N_2 einen Adapter auf Basis von E, d. h., es gibt einen Controller C, sodass $N_1 \oplus E \oplus C \oplus N_2$ das Korrektheitskriterium erfüllt. Da wir portweise komponieren, können wir die Gesamtkomposition als $N_1 \oplus (E \oplus C \oplus N_2)$ schreiben. Da die Komposition immer noch das Korrektheitskriterium erfüllt, ist $(E \oplus C \oplus N_2)$ ein Controller für N_1. Dies steht aber nach Definition 12 eines Controllers im Widerspruch zu Annahme. ∎

Andererseits, wenn für ein offenes Netz N_1 ein Controller existiert, dann schränkt uns die Semantik der Transformationsregeln in der Existenz eines Adapters ein. Wenn wir die Semantik hinter den Transformationsregeln ignorieren und beliebige Transformationsregeln zuließen, können wir immer einen Adapter für N_1 finden, sofern für N_1 ein Controller existiert.

In dem Beweis zu dieser Aussage nutzen wir aus, dass wir für zwei kontrollierbare offene Netze N_1 und N_2 einen trivialen, von unserer Technik unabhängigen Adapter angeben können: Seien C_1 und C_2 Controller für N_1 und N_2, dann erhalten wir einen



(a) Ausgangssituation (b) Adapter

Abbildung 16: Triviale Adaption zweier offener Netze N_1 und N_2 unter Ignorierung der semantischen Bedeutung von Nachrichten.

Adapter A, indem wir C_1 und C_2 parallel komponieren, also $A = C_1 \oplus C_2$. In Komposition mit N_1 und N_2 stellt der Adapter korrektes Verhalten sicher, denn $N_1 \oplus A \oplus N_2 = N_1 \oplus (C_1 \oplus C_2) \oplus N_2$, und da C_1 und C_2 unabhängig voneinander ihre offenen Netze kontrollieren, ist auch die Gesamtkomposition korrekt. Im Beweis lassen wir die beiden Controller über eine spezielle Engine mit N_1 und N_2 interagieren.

Lemma 31 (Existenz eines Adapters hängt nur von Semantik ab)
Für jedes Paar offener Netze N_1 und N_2, für die jeweils ein Controller existiert, existiert ein kanonischer Adapter, der von der semantischen Bedeutung der Transformationsregeln abstrahiert.

Die Beweisidee ist in Abbildung 16 dargestellt. Wir adaptieren die kontrollierbaren offenen Netze N_1 und N_2, indem wir deren Controller nutzen, um eine speziell konstruierte Engine zu kontrollieren.

Wir sehen links in Abbildung 16a die beiden offenen Netze N_1 und N_2 mit ihren Controllern C_1 und C_2. Wir legen Transformationsregeln fest, die die Nachrichten, die N_1 oder N_2 austauschen, erzeugen oder löschen können. Wir nutzen die Controller, um das Löschen und Erzeugen zu steuern, wobei wir jeweils ein kanonisches offenes Netz (C'_1 bzw. C'_2) brauchen, um die Controller mit der Engine komponieren zu können.

Beweis.
Wir führen den Beweis für die Aussage konstruktiv, indem wir die semantischen Bedeutung der Transformationsregeln ignorieren. Wir unterscheiden dafür zwischen synchroner und asynchroner Schnittstelle zwischen Engine und Controller gemäß Definition 24. Wir führen hier den Beweis für den synchronen Fall und erörtern anschließend kurz die Idee für den asynchronen Fall.

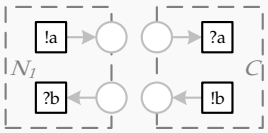


Abbildung 17

Als Ausgangspunkt dienen uns das offene Netz N_1 und dessen Controller C in Abbildung 17. Das offene Netz N_1 repräsentiert ein beliebiges offenes Netz. Die folgende Argumentation ist für jedes Sende- und Empfangslabel in N_1 analog. Wir konstruieren eine spezielle Engine, die über den Controller C gesteuert wird. Wir zeigen mit klassischen Transformationsschritten auf der Netzstruktur, dass wir für beliebige Label auf die ursprüngliche Struktur von $N_1 \oplus C$ zurück kommen. Die konstruierte Engine schränkt somit das Verhalten nicht ein und N_1 und C können immer noch das gleiche Verhalten zeigen.

Für N_2 führen wir die Argumentation analog. Die Engine wird durch die unten angegebene Konstruktion unabhängig für jedes Netz erweitert. Indem wir die einzelnen Controller parallel komponieren, erhalten wir einen gemeinsamen Controller für den Adapter.

Die Idee für einen Regelsatz und somit eine Engine ist, Nachrichten beliebig erzeugen oder löschen zu können. Der gegebene Controller C steuert, wann Nachrichten erzeugt oder gelöscht werden.

Für jedes Empfangslabel $?l$ in N_1 benötigen wir eine Regel R_l: $\rightarrow$ l, die eine Nachricht vom Typ l erzeugen kann, und für jedes Sendelabel $!l$ in N_1 benötigen wir eine Regel R_l: l $\rightarrow$, die eine Nachricht vom Typ l löschen kann. In der Engine erzeugen wir dadurch jeweils zwei Transitionen, die wir synchron kontrollieren. Für unser Beispiel in Abbildung 17 sehen wir eine entsprechende Engine E in Abbildung 18.

Für jedes Label führen wir nun noch eine Controllerkomponente C_l ein, die es uns erlaubt, die synchrone Schnittstelle der Engine asynchron mit Hilfe von C zu steuern. Da jede Transition in der Engine eine synchrone Gegenstelle benötigt, kopieren wir die Struktur aus den beiden Transitionen und dem Platz dazwischen nach C_l und verbinden die Transitionen synchron mit der Engine. Im Falle eines Empfangslabel

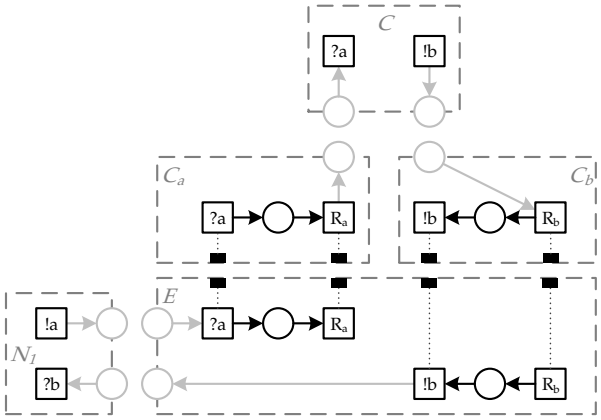


Abbildung 18: Spezielle Engine und Controlleranpassungen, um A zu kontrollieren, wenn die Schnittstelle zum Controller synchron ist.

?l empfängt die Kopie von R_l in C_l die Nachricht vom Typ l von C, im Falle eines Sendelabel !l sendet die Kopie von R_l in C_l die Nachricht vom Typ l an C.

Beim Komponieren verschmelzen quasi die Transitionen in C_l mit ihren Gegenstücken in E. Die Plätze zwischen den beiden Transitionen bilden dann parallele Plätze, die den gleichen Vor- und Nachbereich haben. Nach Murata [80] können wir einen dieser parallelen Plätzen entfernen, ohne die Semantik des Netzes zu ändern. Anschließend können wir über zwei weitere Transformationsschritte auch noch die beiden Transitionen in der Komposition entfernen, weil wir Plätze in Reihe haben. Anschließend ergibt sich genau die Struktur, die sich durch Komposition von N_1 mit C ergeben hätte.

Somit ist für ein offenen Netz N_1 und dessen Controller C, die Komposition von C mit den einzelnen Komponenten C_l und der Engine E ein Adapter. ∎

In Abbildung 19 sehen wir die Beweisidee für den Fall, dass die Schnittstelle zwischen Controller und Engine asynchron ist. Wir definieren analog Regeln R_l für jedes Kommunikationslabel, allerdings müssen die Komponenten C_l etwas anders aussehen.

In diesem Fall nutzen wir aus, dass der Kommunikationsplatz p_2 zwischen ?a in C_a und R_a in E in der Komposition redundant ist, wie man leicht über die Platzinvariante [92] $p_2 = p_a - p_1$ sieht. Wir entfernen diesen Platz, weil er das Verhalten nicht

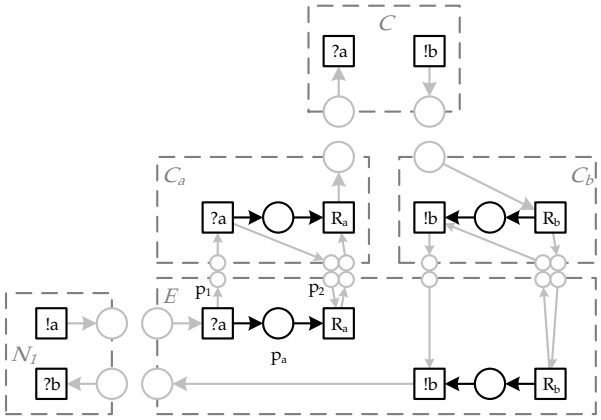


Abbildung 19: Spezielle Engine und Controlleranpassungen, um A zu kontrollieren, wenn die Schnittstelle zum Controller asynchron ist.

beeinflusst. Anschließend entfernen wir die genannten Transitionen, weil wir Plätze in Reihe haben, und die verbleibenden Plätze sind dann wieder parallel. So sind wir bei der gleichen Struktur wie im synchronen Fall und können den Adapter auf den Ausgangsfall in Abbildung 17 zurückführen.

Wir können für zwei kontrollierbare offene Netze N_1 und N_2 eine Art trivialen Adapter generieren. Der Adapter ist insofern trivial, dass wir die Bedeutung von Nachrichten komplett ignorieren und kanonische Transformationsregeln benutzen, die die Nachrichten von N_1 und N_2 löschen oder erzeugen können. Insbesondere besteht so kein Zusammenhang zwischen den Nachrichten. Da wir normalerweise Nachrichten nicht beliebig erzeugen oder löschen können, betrachten wir in Kapitel 7 die Frage, wie wir einem Nutzer helfen können, den Zusammenhang zwischen Nachrichten herzustellen, wenn die Controllersynthese fehlschlägt.

Als nächstes interessiert uns, ob wir die Wahl der Schnittstelle zwischen Engine und Controller eines Adapters Einfluss auf die Existenz eines Adapters hat.

4.2.2 *Wahl der Controllerschnittstelle*

In unserer Definition einer Engine unterscheiden wir explizit zwischen einer synchronen und asynchronen Schnittstelle zwischen Engine und Controller. Die synchrone Schnittstelle hat sich bei der Implementation dieses Ansatz in einem Werkzeug als wesentlich schneller in der Berechnung erwiesen, d. h., Adapter mit einer synchronen Schnittstelle zwischen Engine und Controller lassen sich in der Regel schneller erzeugen als die mit einer asynchronen Schnittstelle. Dies hängt damit zusammen, dass das Senden und Empfangen einer Nachricht nicht in dem Moment geschehen muss, in dem die entsprechende Transition in der Engine sie benötigt oder erzeugt. Es gibt somit viel mehr Situationen, in denen während der Controllersynthese geschaut werden muss, ob eine Nachricht zu senden oder zu empfangen ist, als im synchronen Fall.

Bei Algorithmen erreichen wir oft eine bessere Geschwindigkeit dadurch, dass wir Anforderungen abschwächen. Zum Beispiel kann eine Beschleunigung bedeuten, dass wir nicht mehr in jedem Fall ein positives Ergebnis wie eine Controller erhalten. Jedoch stellen wir fest, dass wir durch die Wahl einer synchronen Schnittstelle nichts einbüßen. Wir finden mindestens dann einen Adapter mit synchroner Schnittstelle zwischen Engine und Controller, wenn wir einen für den asynchronen Fall finden, und dies sogar schneller.

Im Folgenden bezeichnen wir Definition 24, die eine Engine erst einmal ohne Beschränkungen der Plätze einführt, als die *konzeptuelle Definition* einer Engine. Für diese unterscheiden sich die Ergebnisse im Vergleich mit einer Engine, die wir explizit über Komplementärplätze beschränken.

Mooij und Voorhoeve [72] stellen bezüglich der Ausdrucksmächtigkeit von Adaptern fest, dass die konzeptuelle Definition einer Engine ausdrucksmächtiger ist als eine, die explizit eine Beschränkung der Engine über Komplementärplätze erzwingt. Für jede Beschränkung der Engine lässt sich ein Beispiel finden, das für die gewählte Beschränkung in eine Verklemmung gerät, aber korrekt ist, wenn die Beschränkung bezüglich einer höheren Kapazität gewählt wird.

Da die konzeptuelle Engine nicht beschränkt ist, können wir sogar für ein offenes Netz mit unbeschränkten Markierungen und somit unbeschränktem Verhalten einen Adapter angeben – wir können ihn jedoch nicht zwangsläufig ausrechnen. Jede gewählte Kapazität ist dann zu niedrig. Daher schlussfolgern Mooij und Voorhoeve, dass die konzeptuelle Engine ausdrucksmächtiger ist.

Später betrachten Mooij und Voorhoeve [74], ob die Wahl einer asynchronen und synchronen Schnittstelle zwischen Engine zum Controller, wobei die Engine sonst gleich ist, Einfluss auf die Existenz eines Adapters hat. Sie stellen fest, dass bezüglich der konzeptuellen Definition die Wahl der Schnittstelle egal ist. Die Beweisidee geben wir hier kurz an.

Lemma 32 (Unabhängigkeit von der Wahl der Schnittstelle)
Seien N_1 und N_2 zwei offene Netze, $\mathcal{R}$ ein Menge von Transformationsregeln, $E_a = Engine(N_1, N_2, \mathcal{R})$ eine Engine mit asynchroner Schnittstelle und $E_s = Engine(N_1, N_2, \mathcal{R})$ eine Engine mit synchroner Schnittstelle. Dann existiert ein Controller für $N_1 \oplus E_s \oplus N_2$ genau dann, wenn ein Controller für $N_1 \oplus E_a \oplus N_2$ existiert.

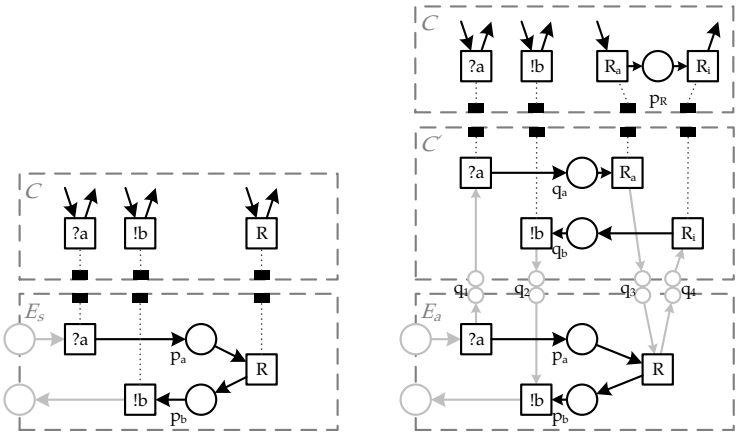
Der Beweis zeigt, dass ein Controller für die asynchrone Engine mit Hilfe einer Übersetzung auch mit der synchronen Engine genutzt werden kann und umgekehrt.

Synchron zu asynchron

Zuerst betrachten wir den Fall einer synchronen Engine E, zu der ein synchroner Controller C gehört, wie in Abbildung 20 zu sehen. Die Engine E_s in Abbildung 20a besteht aus einer Empfangstransition ?a, einer Sendetransition !b und einer Regeltransition R, die jeweils synchron mit dem Controller verbunden sind. Um die Beweisidee zu veranschaulichen, sind wir nicht so streng bei der Benennung von Transitionsnamen und -labeln. Die Transitionen sind entsprechend ihrer Aufgabe in der Engine mit Labeln versehen. Die Transitionen des Controllers greifen diese Label auf, entsprechend der Transition mit der sie synchron kommunizieren.

In Abbildung 20b sehen wir die zu E_s analoge Engine E_a, jedoch jetzt mit einer asynchronen Schnittstelle. Um den synchronen Controller weiter nutzen zu können, erzeugen wir ein offenes Netz C' als Übersetzer zwischen E_a und C. In C' kopieren wir die Struktur aus E_a, jedoch führen wir für jede Regeltransition R eine aktivierende Transition $\mathrm{R_a}$ und eine informierende Transition $\mathrm{R_i}$ ein. Entsprechend teilen wir die Transition R in C in zwei auf gemäß der Murataregeln [80].

Wenn wir $E_a \oplus C' \oplus C$ bilden, dann ist jeder Platz in E_a redundant. So gilt zum Beispiel $p_a = q_1 + q_a + q_3$ für die Anzahl der Marken auf den entsprechenden Plätzen. Somit können wir den Platz $\mathrm{p_a}$ entfernen. Die Redundanz gilt auch für den Platz $\mathrm{p_R}$ in C mit $p_R = q_3 + q_4$, sodass wir auch diesen entfernen können. Was übrig bleibt



(a) synchron (b) asynchron

Abbildung 20: Nutzung eines synchronen Controllers C mit einer asynchronen Engine mit Hilfe der Konstruktion von C'.

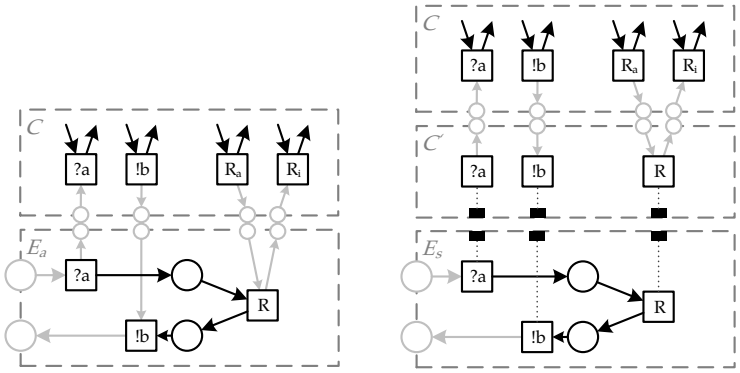
in der Komposition, sind Transitionen, auf die wir die Murataregeln, inbesondere das Zusammenfassen von aufeinander folgenden Transitionen anwenden können. Letztendlich kommen wir zu der Struktur von $E_s \oplus C$, wobei wir q_a mit p_a und q_b mit p_b identifizieren.

Wenn also zu einer synchronen Engine ein Controller existiert, dann existiert auch ein Controller zu der entsprechenden asynchronen Engine.

Asynchron zu synchron

Nun sei E_a eine Engine mit asynchroner Schnittstelle, die genauso wie im vorherigen Beispiel aufgebaut ist, mit C als Controller C. In Abbildung 21 sehen wir, wie wir die analoge synchrone Engine E_s mit dem gleichen Controller nutzen können.

Die Umsetzung der Beweisidee ist hier noch einfacher als im vorherigen Fall. Wie wir in 21b sehen können, definieren wir wieder einen Übersetzer C', der synchron mit der Engine E_s kommuniziert. Abhängig davon, mit welcher Art Transition in E_s eine Transition in C' synchronisiert wird, erhält diese Transition das entsprechende asynchrone Kommunikationsmöglichkeit mit C. Wenn die Transition mit einer Emp-



(a) asynchron (b) synchron

Abbildung 21: Nutzung eines asynchronen Controllers C mit einer synchronen Engine mit Hilfe der Konstruktion von C'.

fangstransition in E_s synchronisiert wird, dann informiert sie C, wenn sie mit einer Sendetransition in E_s synchronisiert wird, dann wird sie von C aktiviert, und wenn sie mit einer Regeltransition synchronisiert wird, dann wird sie sowohl von C aktiviert und informiert C.

Komponieren wir nun E_s mit C, erhalten wir genau E_a. Damit ist klar, dass in Komposition mit C das gleiche Verhalten wie zuvor möglich ist.

Wenn also zu einer synchronen Engine ein Controller existiert, dann existiert auch ein Controller zu der entsprechenden asynchronen Engine.

Auswirkungen der Controllerschnittstelle auf beschränkte Netze

Wir haben gesehen, dass es konzeptuell keinen Unterschied macht, ob wir uns für eine synchrone oder asynchrone Schnittstelle zwischen Engine und Controller entscheiden. Allerdings haben wir im vorherigen Kapitel bereits festgestellt, dass wir die Engine beschränken müssen, um einen Controller tatsächlich ausrechnen zu können.

Im Fall einer beschränkten Engine, genauer einer Beschränkung mit Komplementärplätzen, stellen Mooij und Voorhoeve [74] fest, dass eine Engine mit synchroner Schnittstelle ausdrucksmächtiger ist. Das heißt, es gibt Fälle, in denen wir für eine Engi-

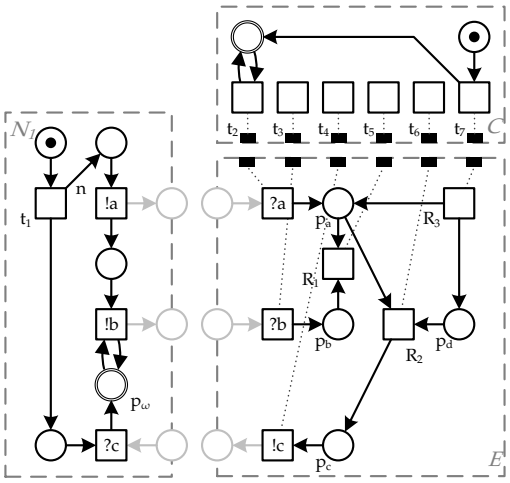


Abbildung 22: Beispiel für eine synchron, aber nicht asynchron kontrollierbare Engine mit Kapazitäten

ne mit synchroner Schnittstelle einen Controller finden können, aber mit asynchroner Schnittstelle nicht. Ein entsprechendes Beispiel sehen wir in Abbildung 22.

Das gegebene offene Netz N_1 erzeugt über die Transition t_1 eine Menge von n Marken – wobei n eine beliebige, natürliche Zahl größer 0 sei –, die sukzessive konsumiert werden, indem die Transition !a eine Nachricht a sendet. Sobald N_1 eine Nachricht c über die Transition ?c empfängt, kann N_1 für jedes gesendete a ein b über die Transition !b senden. Das offene Netz N_1 terminiert, wenn es nach dem Empfang von c insgesamt n Nachrichten vom Typ b gesendet hat.

Die Engine E implementiert drei Transformationsregeln: R_1: a, b $\rightarrow$, R_2: a, d $\rightarrow$ c und R_3: $\rightarrow$ a, d, wobei a, b und c Nachrichten sind, die N_1 und E austauschen, und d eine Nachricht ist, die ausschließlich von der Engine benutzt wird. Die erste Regel R_1 konsumiert jeweils ein a und ein b. Die zweite Regel R_2 erzeugt die Nachricht c, die notwendig ist, damit N_1 überhaupt eine Nachricht vom Typ b senden kann. Und die dritte Regel R_3 ermöglicht, dass R_2 angewendet werden kann, indem je eine Nachricht a und d erzeugt wird.

Der abgebildete Controller C schaltet einmalig die Transition t_7, sodass die Regel R_3 angewendet wird. Anschließend darf die Transition t_2 beliebig schalten, und somit

kann E Nachrichten von Typ a empfangen. Alle anderen Transitionen $t_3, \ldots t_6$ dürfen jederzeit schalten. Die Regeln in der Engine können nur in der Reihenfolge einmal R_3, einmal R_2 und dann n-mal R_1 schalten. Nach dem Schalten von R_3 kann E immer wieder Nachrichten vom Typ a empfangen. Nachdem R_2 und anschließend !c geschaltet haben, kann E auch immer wieder Nachrichten vom Typ b empfangen. Über die Transition R_1 werden die Nachrichten vom Typ a und b dann entfernt. Das Gesamtsystem $N_1 \oplus E \oplus C$ terminiert somit schwach.

Das gleiche Verhalten ist immer noch möglich, wenn wir Kapazitäten für die Plätze von E mit Hilfe von Komplementärplätzen erzwingen, jedoch nicht mehr, wenn wir stattdessen eine asynchrone Schnittstelle zwischen Engine und Controller benutzen. Folgende Überlegung zeigt, dass wir im asynchronen Fall die Kapazität ungünstig wählen können. Sei $k \leq$ n eine Kapazität, die für jeden Platz in E gelten soll. Nun gibt es den Ablauf, in dem N_1 seine n Nachrichten vom Typ a an E sendet. Weder E noch C können im asynchronen Fall verhindern, dass E diese Nachrichten empfängt. Der Controller C wird über den Empfang nur informiert und kann die Empfangstransition nicht weiter steuern. Deshalb gibt es einen Ablauf, in dem k Marken auf p_a produziert werden. Nun kann R_3 nicht mehr angewendet werden, weil der Komplementärplatz zu p_a leer ist. Damit kann R_2 nicht schalten, weil wir die Nachricht d nicht erzeugen können. Folglich kann auch R_1 nicht schalten, weil keine Nachricht vom Typ b an E gesendet werden kann.

In ihrem Beispiel benutzen Mooij und Veerhoeve sogar ein N_1, das unbeschränkt viele Nachrichten vom Typ a und b senden kann, jedoch immer gleich viele von beiden Typen. Sie argumentieren, dass das geschilderte Problem somit für jede Kapazität k auftritt. Daraus folgt, dass egal welche Kapazität wir für die Plätze wählen, gibt es immer ein Beispiel, für das wir ein Adapter mit synchroner Schnittstelle zwischen Engine und Controller finden, aber nicht mit asynchroner.

Lemma 33 (Controllerschnittstelle bei beschränkten Netzen)
Seien N_1 und N_2 zwei offene Netze, $\mathcal{R}$ ein Menge von Transformationsregeln, $E_a = \mathit{Engine}(N_1, N_2, \mathcal{R})$ eine beschränkte Engine mit asynchroner Schnittstelle und $E_s = \mathit{Engine}(N_1, N_2, \mathcal{R})$ eine beschränkte Engine mit synchroner Schnittstelle. Dann existiert ein Controller für $N_1 \oplus E_s \oplus N_2$, wenn ein Controller für $N_1 \oplus E_a \oplus N_2$ existiert. Die Rückrichtung gilt nicht.

Der Beweis der Implikation ist analog zu Lemma 32. Dass die Rückrichtung nicht gilt, zeigt das obige Beispiel.

Soweit sind die gemachten Schlussfolgerungen nachvollziehbar, jedoch lässt sich diese Argumentation nicht auf die Algorithmen zur Controllersynthese übertragen, die wir verwenden. Diese Algorithmen funktionieren nur, wenn das zu kontrollierende offenen Netz beschränkt ist. Dies betrifft auch die Kommunikationsplätze zwischen N_1 und E, die im Beispiel von Mooij und Veerhoeve beliebig viele Marken enthalten können. Wenn wir offene Netze mit endlichem Verhalten unter Berücksichtigung einer vorgegebenen Kommunikationsschranke betrachten, dann können wir eine Schranke k für das komponierte Gesamtsystem ermitteln, für die die Äquivalenz der beiden Schnittstellenarten gilt, da wir die Kapazität dann entsprechend auf k setzen können.

Wir können also feststellen, dass die Existenz eines Adapters nicht von der Wahl einer synchronen und das asynchronen Schnittstelle in der konzeptuellen 24 abhängt. Wenn wir jedoch eine Beschränkung durch eine Kapazität für die Plätze der Engine einführen, existieren Fälle, in denen wir eine Adapter mit synchroner Schnittstelle, aber nicht mit asynchroner Schnittstelle zwischen Engine und Controller synthetisieren können.

4.3 Semantik der Engine

In diesem Abschnitt betrachten wir, welche Möglichkeit die Engine bietet, um Systeme zu modellieren, und wie wir die Semantik der Engine unseren Bedürfnissen weiter anpassen können. Als Beispiel dient uns die Betrachtung eines *Firewallsystems*.

Mit einer Engine können wir ganz allgemein den Datenfluss zwischen zwei offenen Systemen beschreiben. Den Datenfluss können wir dabei beliebig strukturieren; wir können ihm mit Hilfe der Transformationsregeln sogar jede beliebige Petrinetzstruktur geben. Insofern lässt sich das Konzept einer Engine auf viele Arten offener Systeme übertragen und der Datenfluss entsprechend strukturieren.

Das Kommunikationsmodell beziehungsweise die Kommunikationssemantik kann sich jeodch bei verschiedenen offenen Systemen unterscheiden. Wir betrachten in dieser Arbeit zum Beispiel asynchronen Nachrichtenaustausch, in dem sich Nachrichten überholen können. Viele real existierende System basieren auf asynchroner Kommunikation über Warteschlangen. Diese unterschiedlichen Kommunikationssemantiken

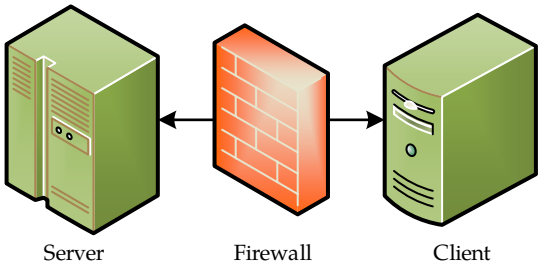


Abbildung 23: Firewall zum Prüfen und potentiellen Löschen von Nachrichten zwischen Server und Client

können wir nicht über Transformationsregeln modellieren – was auch nicht deren Sinn wäre – wir müssen die Semantik der Engine explizit anpassen.

Das Verhalten eines offenes Netzes hängt neben der Struktur von seiner Anfangsmarkierung und seinen Endmarkierungen ab. Diese können wir nicht einfach in die Engine abbilden, da die Anfangs- und einzige Endmarkierung die leere Markierung ist. Wenn wir zum Beispiel die Engine beschränken wollen wie in Abschnitt 3.4, was eine Anfangsmarkierung der Komplementärplätze zur Folge hat, müssen wir dies explizit definieren.

Um die Semantik einer Engine anzupassen, können wir eine beliebige Struktur über Transformationsregeln vorgeben, die über weitere Definitionen verfeinert werden. Mit den bisherigen Ausdrucksmitteln können zudem die Interaktion der Engine mit den gegebenen offenen Netzen nicht beeinflussen.

Deshalb modellieren wir den gewünschten Nachrichtenfluss zwischen Systemen mit Transformationsregeln aufbauend auf der Idee der Engine. Die Semantik der Engine passen wir anschließend weiter an, damit sie dem Anwendungsfall entspricht.

4.3.1 *Aufgaben einer Firewall*

Eine *Firewall* ist Teil eines Netzwerkes und unterscheidet zwischen zwei Seiten: innen und außen. Allgemein regelt eine Firewall, ob Nachrichten von außen nach innen oder von innen nach außen gelangen dürfen. Falls eine Nachricht valide ist, wird sie hindurchgelassen, anderenfalls wird die Nachricht gelöscht.

Vereinfacht nehmen wir an, dass die Firewall zwischen einem Server innen und einem Client außen sitzt, wie in Abbildung 23 abgebildet. Im allgemeinen schützt eine Firewall den Server vor unberechtigten Zugriffen, ohne dabei berechtigte Zugriffe zu beeinträchtigen. Auf dem Server laufen zum Beispiel verschiedene Dienste, von denen einer nach außen zur Verfügung steht, die anderen jedoch nicht von außen benutzt werden sollen. Ein Client möchte den verfügbaren Dienst nutzen. Die Firewall ermöglicht es dem Client, den Dienst zu benutzen, und schützt den Server vor unberechtigten weiteren Zugriffen. Der Client darf nicht widerrechtlich die anderen Dienste nutzen.

In einem typischen Firewallsystem gibt ein Benutzer Regeln darüber an, welche Nachrichten durchgelassen und welche gelöscht werden. Im Bereich der *Stateful Packet Inspection*, der zustandsbasierten Überprüfung von Nachrichtenpaketen, wird dabei sogar unterschieden, ob eine Nachricht zu einer bestehenden Verbindung gehört oder nicht. Es gibt also zwei Zustände, einen für eine *neue* Verbindung, und einen für eine *bestehende* Verbindung. Abhängig von diesen Zuständen kann der Benutzer die Regeln unterschiedlich definieren. Wurde eine Verbindung erst einmal korrekt aufgebaut, werden in der Regel andere Arten von Nachrichten ausgetauscht, als bei einer neuen Verbindung.

Bei dem weit verbreiteten System *iptables* [46] gibt es genau diese zwei Zustände, die eine Verbindung haben kann. Verbindungsprotokolle sind in der Regel wesentlich komplexer; allein für den Aufbau einer Verbindung müssen mehrere Nachrichten ausgetauscht werden und es wird jeweils ein neuer Zustand in den entsprechenden Protokollen erreicht. Auch im Stadium des Aufbaus einer Verbindung ist kritisch, ob eine Nachricht korrekter Weise ausgetauscht wird. Komplexere Protokolle können auch eine weitere Verfeinerung des Zustandes, in dem eine Verbindung bereits besteht, erfordern. Bei *iptables* ist dies nicht möglich, dort wird für die Existenz einer Verbindung auf einen Zustand abstrahiert. Jede Nachricht, die bei bestehender Verbindung ausgetauscht werden muss, wird gleich berechtigt betrachtet, auch wenn das Protokoll bestimmte Nachrichten nur in einzelnen Zuständen austauscht. Bestimmte Reihenfolgen oder andere Abhängigkeiten zwischen Nachrichten können wir nicht ausdrücken.

Da wir mit *iptables* nur zwei Zustände unterscheiden können, übertragen wir die Idee von Engine und Controller auf das Konzept einer Firewall. Die Controllersynthese versetzt uns in die Lage, alle Zustände eines Protokolls zu unterscheiden. Die Engine

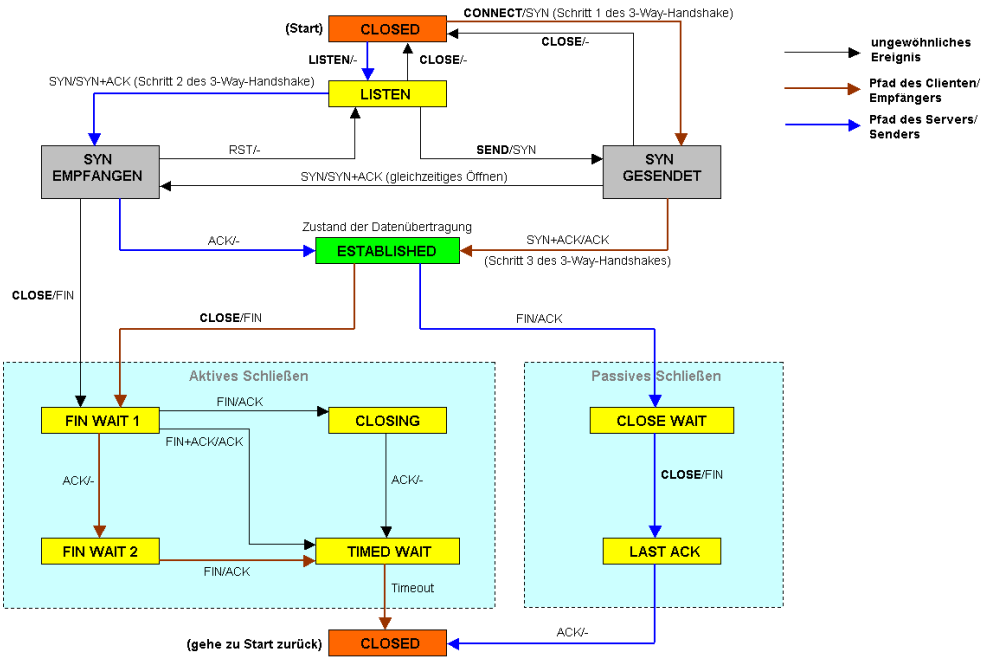


Abbildung 24: Transmission Control Protocol[1]

setzt lediglich die Filterung einer Nachricht um, entweder durch Weiterleiten oder Löschen der Nachricht.

Am Beispiel des *Transmission Control Protocol* sehen wir, dass wir das Verhalten in einer Firewall, die als Adapter umgesetzt ist, viel detaillierter betrachten können.

4.3.2 *Transmission Control Protocol*

Das *Transmission Control Protocol* (TCP) wird häufig genutzt, um Verbindungen zwischen zwei Systemen aufzubauen. Es gibt vor, wie Systeme eine Verbindung aufbauen, nutzen und wieder schließen. Eine vereinfachte Form des Protokolls sehen wir in Abbildung 24. Es beinhaltet das Verhalten sowohl für die Seite, die die Verbindung initiiert (blau), als auch für die Seite, zu der die Verbindung hergestellt wird (braun).

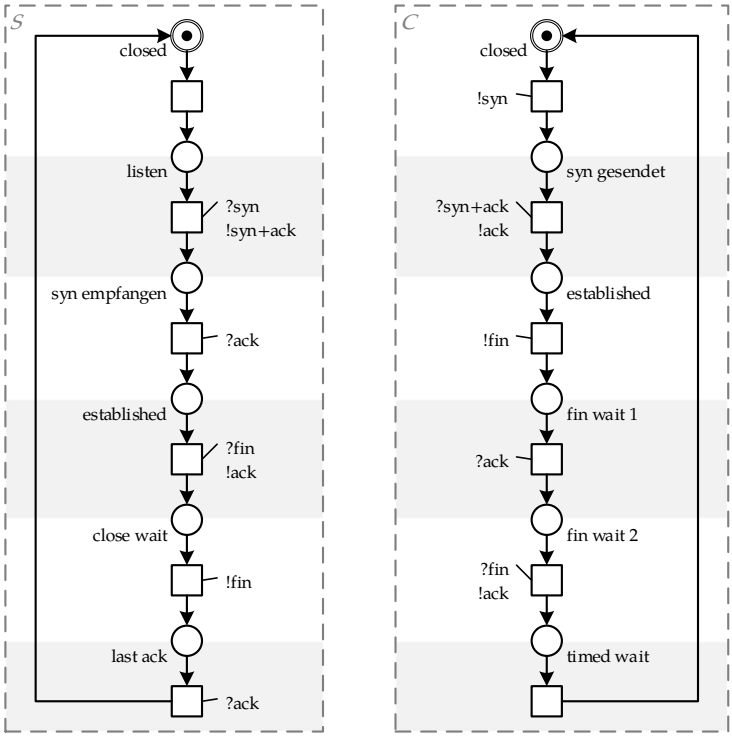
Die Rechtecke stellen die Zustände des Protokolls dar. Der Zustand CLOSED ist der Anfangszustand. Ein Pfeil gibt einen Zustandsübergang an, wobei die Notation der Pfeilannotation bedeutet: „auslösendes Ereignis / Effekt“. Das auslösende Ereignis kann ein aktiver, selbstgewählter Übergang sein, wie zum Beispiel bei **LISTEN**/– vom Zustand CLOSED zum Zustand LISTEN, und wird dann fett geschrieben, oder es ist eine Nachricht, wie zum Beispiel bei SYN/SYN + ACK vom Zustand LISTEN zum Zustand SYN EMPFANGEN. Der Effekt ist entweder eine Nachricht, oder es gibt keinen Effekt, was durch „–“ ausgedrückt wird.

Die hier betrachteten Nachrichten beschreiben auf technischer Ebene gesetzte Bits im Kopf einer Nachricht. Die Nachricht SYN bedeutet also eine Nachricht, bei der das Bit bei SYN gesetzt ist. Entsprechend bedeutet SYN + ACK, dass die Bits bei SYN und ACK gesetzt sind. Da es aber jeweils eine Nachricht ist, teilen wir die Nachrichten gemäß ihrer gesetzten Bits in Typen ein. Von den weiteren Anforderungen an eine Nachricht zum Aufbau einer Verbindung über TCP abstrahieren wir an dieser Stelle, da sie im Wesentlichen weitere Einschränkungen darstellen.

Mit Hilfe des dargestellten Verhaltens wollen wir das Verhalten eines Servers S und eines Clients C als offene Netze modellieren. Das Ergebnis sehen wir in Abbildung 25. Ein Client baut eine Verbindung zum Server auf. Um die Übersichtlichkeit zu verbessern, befinden sich die Kommunikationslabel diesmal nicht in den Transitionen, sondern als Annotationen mit einem Strich mit den Transitionen verbunden. Die einzelnen Zustandsübergänge werden farblich unterschieden.

Bei der Notation bleiben wir möglichst nah an der Nomenklatur von TCP. So enthalten beide Netze den Nachrichtentyp ack sowohl in sendender als auch in empfangender Richtung. Dies ist laut unserer Definition einer Schnittstelle eines offenen Netzes nicht möglich. Allerdings lassen wir es hier zu, da die Richtung eindeutig definiert ist. In einer praktischen Umsetzung lassen sich ein- und ausgehenden Nachrichten vom gleichen Typ zum Beispiel durch einen Präfix unterscheiden.

1 „Tcp verbindung“ von Benutzer:Appaloosa - selbst erstellt// abgemalt aus Tanenbaum - Computer Networks (4th Edition) // Grundlegende Regeln des Zitierens sollten beachtet werden finde ich...Anmerkung: Muß nicht vom Tanenbaum stammen. Kann auch aus den RFCs übernommen sein. Denn in RFC 793 (Seite 23) ist das Bild als ASCII-Grafik eingebunden.. Lizenziert unter Creative Commons Attribution-Share Alike 3.0 über Wikimedia Commons - http://tinyurl.com/nkllzy2



(a) Server *S* (b) Client *C*

Abbildung 25: Server und Client mit der Umsetzung von TCP als offene Netze

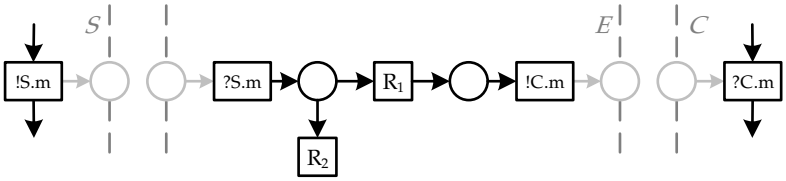


Abbildung 26: Teil der Engine, welcher die Möglichkeiten einer Firewall für den Nachrichtentyp *m* nachbildet

4.3.3 *Aufgaben der Engine*

Die Engine soll die Rolle der Firewall übernehmen, d. h., Nachrichten durchzulassen oder zu löschen. Dies drücken wir mit Transformationsregeln aus, je Nachrichtentyp und Richtungen gibt es genau zwei Regeln: Die erste leitet die Nachricht weiter, die zweite löscht sie. Für eine Nachricht *m*, welche die Typen ack, fin, syn oder eine Kombination dieser Typen repräsentiert, sehen wir die Umsetzung der Engine in Abbildung 26.

Der Server *S* sendet die Nachricht *m*, die von der Engine *E* empfangen wird. Die erste Regel R_1: S.m → C.m modelliert die Weiterleitung von *S* zum Client *C*, der diese dann empfangen kann. Die zweite Regel R_2: S.m → modelliert das Löschen der Nachricht.

Mit der Angabe der Transformationsregeln für jeden möglichen Nachrichtentyp ergibt sich bereits eine Engine *E*. Allerdings verhält sich die Engine noch untypisch für ein Firewallsystem. In *E* können sich mehrere Nachrichten gleichzeitig befinden, in beide Richtungen, und somit sich auch Nachrichten überholen. Für die meisten Protokolle ist aber das strikte Bewahren der Reihenfolge notwendig. Wir schränken daher das mögliche Verhalten weiter ein.

4.3.4 *Semantik der Engine anpassen*

Wir passen nun die Semantik der Engine so an, dass sie mit der Kommunikationssemantik typischer Firewallsysteme übereinstimmt. Der so generierte Controller ist dann entsprechend auf diese Semantik abgestimmt und kann in existierenden Systemen eingesetzt werden, um das Filtern von Nachrichten zu steuern.

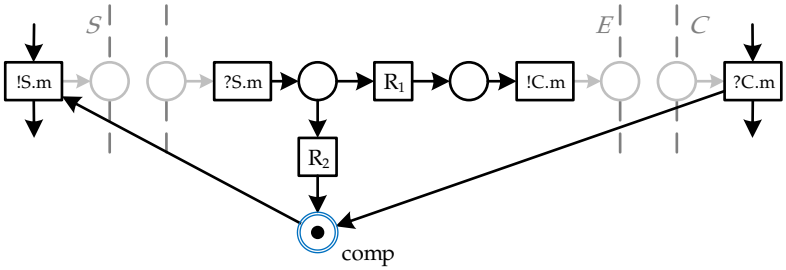


Abbildung 27: Engine aus Abbildung 26 erweitert um einen Komplementärplatz comp zum Einschränken der Kommunikation.

Im konkreten Beispiel müssen wir zwei Punkte beachten: Die Nachrichten werden von der Firewall in der Reihenfolge abgearbeitet, in der die Firewall diese erhält, und Nachrichten überholen sich bei bei TCP nicht.

In unserem Modell ist es möglich, dass der Client, wenn er im Zustand syn gesendet ist und zu established übergeht, die Nachricht !ack sendet und anschließend direkt die Nachricht !fin. In unserem asynchronen Modell können sich diese beiden Nachrichten überholen, d. h., die Nachricht fin könnte vor der Nachricht ack von der Engine empfangen werden, obwohl die Nachrichten in einer anderen Reihenfolge gesendet wurden.

Wir wollen erreichen, dass Nachrichten genau in der Reihenfolge von der Engine abgearbeitet werden, in der sie gesendet wurden. Eine mögliche Umsetzung ist in Abbildung 27 zu sehen.

Wir fügen ein globales Semaphor ein, über das eine Transition nur dann aktiviert wird, wenn keine Nachricht in der Engine vorliegt, und die Firewall somit keine Entscheidung treffen muss. Das Semaphor liegt initial auf dem Platz comp, mit dem alle sendenden und empfangenden Transitionen von S und C verbunden sind. Wenn die Nachricht m über !S.m gesendet wird, dann entfernt diese Transition das Semaphor. Es wird zurückgegeben, wenn die Transition ?C.m die Nachricht empfängt, oder wenn die Nachricht durch R_2 gelöscht wird.

4.3.5 *Erkenntnisse*

Wenn wir für die gegebenen offenen Netze für Server S und Client C und die angepasste Engine E einen Controller synthetisieren, stellen wir fest, dass niemals die Regel zum Löschen einer Nachricht angewendet wird. Dies ist auch sinnvoll, da jede der gesendeten Nachrichten notwendig im TCP ist. Da weder S noch C eine Nachricht erneut senden, bedeutet das Löschen einer Nachricht durch die Engine, dass das TCP nicht mehr korrekt befolgt werden kann.

Wir haben für Server und Client genau das korrekte Verhalten angegeben, wie wir es uns in der Interaktion wünschen. Der Controller beschreibt also alle positiven Fälle, in denen eine Nachricht durchgelassen werden darf. Wenn wir den Controller nutzen wollen, um tatsächlich eine Firewall für diese Art von Server und Client umzusetzen, muss die Firewall in der Lage sein, die betrachteten Nachrichten zu unterscheiden und die Möglichkeit bieten, Nachrichten zu löschen oder durchzulassen. Der Controller nutzen wir, um zu entscheiden, was mit einer Nachricht geschieht. Wenn eine Nachricht ankommt, unterscheidet die Firewall, ob im momentanen Zustand des Controllers ein Übergang für die Nachricht vorhanden ist. Falls ja, wird die Nachricht durchgelassen und der Controller wechselt in den entsprechenden Nachfolgezustand; falls nein, wird die Nachricht gelöscht.

Die Idee einer solchen Art Firewall ist also nicht mehr, manuell Filterregeln anzugeben, sondern das korrekte Verhalten der Kommunikationspartner zu spezifizieren und einen Controller zu ermitteln, der genau die Nachricht durchlässt, die zum korrekten Kommunikationsverhalten passen.

In unserem Beispiel ist die Anzahl der Zustände im Vergleich zu einer Firewallimplementation wie iptables sehr groß ist. Für jede Nachricht existiert ein Zustand für das Empfangen der Nachricht, für die Anwendung der Transformationsregel und schließlich für das Senden. Da im betrachteten Beispiel sieben Nachrichten ausgetauscht werden, ergibt sich eine Zyklus mit insgesamt 21 Zuständen ($1 \xrightarrow{?C.syn} 2 \xrightarrow{R_1} 3 \xrightarrow{!S.syn} 4 \xrightarrow{?S.syn+ack} \ldots \xrightarrow{?S.fin} 17 \xrightarrow{R_1} 18 \xrightarrow{!C.fin} 19 \xrightarrow{?C.ack} 20 \xrightarrow{R_1} 21 \xrightarrow{!S.ack} 1$).

Es fällt auf, dass in dem betrachteten Beispiel die Regeln zum Löschen von Nachrichten nicht benutzt werden. Wie beschrieben, ist das in diesem Fall sinnvoll, weil alle Nachrichten für die korrekte Interaktion benötigt werden. Die Regel zum Löschen von Nachrichten wird interessant, wenn wir gezielt fehlerhaftes Verhalten modellieren.

Die Firewall soll am Ende vielleicht gar nicht alle Nachrichten löschen, sondern nur solche, die einen Angriff auf eine zu schützenden Dienst darstellen.

Zusammenfassend ermöglicht die Idee einer Engine, den Nachrichtenfluss zwischen zwei kommunizierenden Systemen zu beschreiben. Weiteren allgemeine strukturelle Änderungen der Engine ändern deren Semantik soweit, dass wir ein Nachrichten austauschendes System sinnvoll nachbilden können.

Mit Hilfe der Controllersynthese können wir die Steuerung der Engine automatisch erzeugen. Der Controller lässt sich in einem existierenden System einsetzen, um abhängig von den Zuständen der beteiligten Systeme den Nachrichtenfluss zu steuern. Dabei enthält der Controller genau so viele Zustände, um die Zustände der Systeme bestmöglich zu unterscheiden.

Die Trennung eines Adapters in Daten- und Kontrollfluss erweist sich hier als vorteilhaft. Die Transformationsregeln können wir automatisch bestimmen und erhalten so die Engine. Die Controllersynthese übernimmt die Aufgabe, Firewallregeln manuell für ein bestimmtes Protokoll angeben zu müssen.

4.4 BEEINFLUSSUNG DER CONTROLLERSYNTHESE

Für die von uns genutzten Techniken zur Controllersynthese gibt es zahlreiche Erweiterungen, die es erlauben, das endgültige Ergebnis eines Adapters zu beeinflussen. Dabei geht es um Einschränkungen beziehungsweise Optimierungen im Verhalten.

Wir zeigen zuerst, welche Möglichkeiten wir haben, das Schalten von Transitionen zu erzwingen, zu verbieten oder partiell zu erzwingen. Für diese Möglichkeiten beschreiben wir, welche Bedeutung sie für die Anwendung auf Transformationsregeln oder die Adaptersynthese haben.

Anschließend stellen wir eine Möglichkeit vor, allgemeinste Kommunikationspartner zu optimieren. So schränken wir die Existenz eines Adapter im Voraus nicht ein, wie es beim Verbieten von Transitionen möglich ist, sondern entfernen im Nachhinein unerwünschtes Verhalten, wenn es die Korrektheit des Adapters nicht verletzt.

4.4.1 *Verhaltenseinschränkungen*

In Definition 19 definieren wir einen allgemeinsten Kommunikationspartner. Dieser lässt sich durch Annotationen zu einer *Bedienungsanleitung* erweitern [61, 112]. Eine Bedienungsanleitung ist eine Charakterisierung aller offenen Netze, die mit einem gegebenen offenen Netz korrekt kommunizieren.

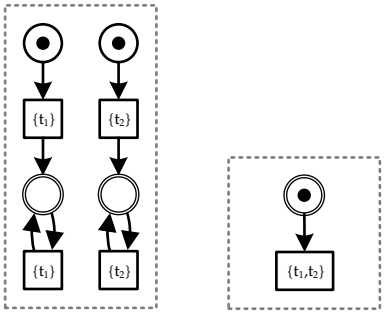
Aufbauend auf dem Konzept der Bedienungsanleitung entwickeln Lohmann, Massuthe und Wolf [60] ein Konzept, um das Schalten von Transitionen eines offenen Netzes zu erzwingen beziehungsweise auszuschließen. Stahl und Wolf [100] beschreiben eine Möglichkeit, Transitionen eines offenen Netzes zu überdecken, d. h., in wenigstens einem Ablauf muss eine zu überdeckende Transition schalten.

Erzwingen und Verhindern von Transitionen

Die Idee, bestimmte Transition zu erzwingen beziehungsweise zu verhindern, entstand im Zusammenhang der Partnersynthese [60]. Der Begriff *Partner* ist hier synonym mit Controller zu verstehen. Da wir mit einer Bedienungsanleitung alle korrekten Partner eines offenen Netzes charakterisieren, soll eine angepasste Bedienungsanleitung alle korrekten Partner charakterisieren, die zwangsläufig eine bestimmte Transition schalten oder niemals schalten. Ein Beispiel hierfür sind unterschiedliche Zahlungsmethoden in einem Onlineshop: Wir wollen wissen, wie die Partner aussehen, die zum Beispiel immer per Lastschrift beziehungsweise niemals per Kreditkarte zahlen.

Als Methode zur Beschreibung einer solchen Einschränkung benutzen wir spezielle, gelabelte Petrinetze, die *Einschränkungsnetze*. Ein entsprechendes Beispiel sehen wir in Abbildung 28. Die gezeigten Einschränkungsnetze wenden wir auf ein Netz N an.

Das Einschränkungsnetz N_E in Abbildung 28a erzwingt das Schalten der beiden Transitionen t_1 und t_2 in einem offenen Netz N. Eine in N_E aktivierte Transition t schaltet genau dann, wenn im zugehörigen Netz N eine Transition t_i schaltet, die in der Labelmenge von t enthalten ist. Damit N_E die Endmarkierung auf den doppelt umrandeten Plätzen erreicht, müssen also die initial aktivierten Transitionen mit den Labeln $\{\mathrm{t}_1\}$ und $\{\mathrm{t}_2\}$ schalten, und somit die Transitionen t_1 und t_2 in N. Mit jedem weiteren Schalten von t_1 oder t_2 in N schaltet die entsprechende Transition in N_E, die wieder zur Endmarkierung führt. Die beiden unteren Transitionen in N_E sind wichtig, weil sonst t_1 und t_2 in N nur genau einmal schalten dürften. Würden wir statt der



(a) Erzwingen (b) Vermeiden

Abbildung 28: Spezielle Petrinetze zum Erzwingen und Vermeiden einer Transition

zwei Stränge nur einen Strang mit den Labeln $\{t_1, t_2\}$ angeben, hieße dies, dass wir das Schalten wenigstens einer der beiden Transitionen t_1 und t_2 erzwingen wollten. Wenn wir N_E als Spezifikation zu N dazu nehmen, bedeutet dies also, dass in jedem Ablauf von N, die Transitionen t_1 und t_2 jeweils mindestens einmal schalten müssen.

Das Einschränkungsnetz N_E in Abbildung 28b vermeidet das Schalten der beiden Transitionen t_1 und t_2 in einem offenen Netz N. In N_E ist die initiale Markierung gleichzeitig die gewünschte Endmarkierung. Sobald die Transition schaltet, weil t_1 oder t_2 in N schalten, verlassen wir die Endmarkierung von N_E und können sie nicht mehr erreichen. Somit beeinflusst N_E die Spezifikation von N soweit, dass t_1 und t_2 in keinem Ablauf schalten dürfen, da wir sonst keine Endmarkierung erreichen können.

Die Nutzung dieser Einschränkungsnetze erfolgt durch Transitionsverschmelzung mit einem gegebenen offenen Netz N. Je eine Kopie der Transition t eines Einschränkungsnetzes, also eine neue Transition t' mit dem gleichen Vor- und Nachbereich wie t, wird dabei mit einer Transition von N verschmolzen, die in der Labelmenge von t steht. Die Transition t wird abschließend entfernt.

Für sich genommen, haben die beiden Einschränkungsnetze folgende Bedeutung für die Adaptersynthese.

VERMEIDEN VON TRANSITIONEN Es ist naheliegend Transitionen zu vermeiden, welche die Transformationsregeln umsetzen. Die Transformationsregeln sind es, die einen Adapter maßgeblich beeinflussen. Eine Regeltransition zu vermeiden hat den

gleichen Effekt wie dessen entsprechende Transformationsregel aus der Menge der Transformationsregeln zu entfernen. Andererseits wir jedoch beliebige Transitionen vermeiden. Wir zwingen den Controller zum Beispiel dazu, dass eine bestimmte Nachricht von der Engine nicht empfangen oder nicht gesendet wird. Alternativ lässt sich so auch Verhalten direkt in einem der gegebenen offenen Netze ausschließen, zum Beispiel aufgrund vertraglicher Festlegungen. Ein Beispiel dafür ist, dass nicht mit Kreditkarte bezahlt werden soll. Dafür schließen wir entweder eine entsprechende Transition im Onlineshop aus, oder wir vermeiden es, die Daten mit der Engine zu senden oder zu empfangen. Teile des Verhaltens auszuschließen hat natürlich Auswirkungen auf das Gesamtverhalten. Wenn die Engine eine bestimmte Nachricht nicht empfangen darf, darf das entsprechende offene Netz gar nicht erst senden, um die Korrektheit nicht zu verletzen. Der Controller muss also den Nachrichtenfluss so steuern, dass das unerwünschte Verhalten nicht eintreten kann. Es ist möglich, dass kein solcher Controller existiert.

ERZWINGEN VON TRANSITIONEN Analog zum Vermeiden von Transitionen können wir sowohl Transitionen in der Engine als auch Transitionen in den gegebenen offenen Netzen erzwingen. Im Falle eines gegebenen offenen Netzes wollen wir zum Beispiel für einen Onlineshops erzwingen, dass mit Lastschrift gezahlt wird. Eine Transformationsregel zu erzwingen, impliziert letztendlich Verhalten, dass in beiden offenen Netzen eintreten muss. Anstelle zu fordern, dass der Onlineshop Rechnung und Versandbestätigung schicken muss, und dass der Kunde seinen Bestellabschluss erhält, erzwingen wir die Transformationsregel, die Rechnung und Versandbestätigung des Onlineshops zum Bestellabschluss zusammenfasst.

KOMPLEXERE NETZE Wir können uns neben den beiden Arten von Einschränkungsnetzen in Abbildung 28 auch komplexere Netze vorstellen. So können wir die Reihenfolge von Transitionen vorgeben, sodass eine Transformationsregel vor einer anderen angewendet werden muss, oder spezieller, nur mit Kreditkarte gezahlt werden kann, wenn ein Kunde sich zuvor angemeldet hat. Lohmann et al. setzen diese komplexeren Szenarien mit einer Automatennotation um, die wir aber in Einschränkungsnetze übersetzen können. Bis zu einem gewissen Grad können wir komplexere Netze direkt mit Transformationsregeln nachbilden. Um die beiden Transformationsregeln $R_1 \colon X_1 \to Y_1$ und $R_2 \colon X_2 \to Y_2$ in diese Reihenfolge zu bringen, nutzen wir eine neues,

eindeutiges z, das von der ersten Regel erzeugt und von der zweiten Regel benutzt wird: $R_1\colon X_1 \to Y_1, z$ und $R_2\colon z, X_2 \to Y_2$. Somit kann die zweite Transformationsregel erst nach der ersten Transformationsregel angewendet werden. Allerdings erzwingen wir so weder, dass die erste Regel überhaupt ausgeführt wird, noch können wir Transformationsregeln so angeben, dass wir eine bestimmte Reihenfolge ausschließen.

Die Einschränkungsnetze ermöglichen somit, das Verhalten eines Adapters durch Einschränkungen zu ergänzen.

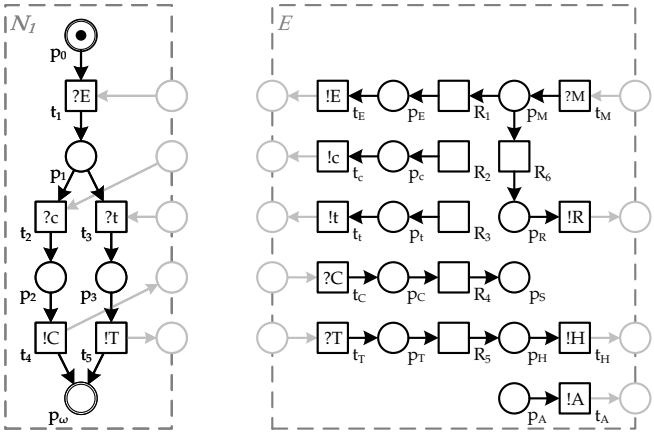
Überdecken von Transitionen

Stahl und Wolf [100] stellen eine Methode vor, um Transitionen und Plätze zu überdecken. Als wichtigste Motivation dient ihnen die Soundness von Prozessen [4], und hierbei insbesondere die Eigenschaft, dass keine Transition eines offenen Netzes tot ist, also in keinem Ablauf schalten kann.

Analog zum obigen Ansatz, können wir diesen Ansatz auch für die Synthese von Adaptern nutzen. Ein Adapter kann formal korrekt sein, und trotzdem kein erwünschtes Verhalten erlauben. So wollen wir zum Beispiel, dass ein Adapter in der Lage sein muss, eine Transformationsregel zur Zahlungsabwicklung tatsächlich auszuführen und nicht nur Bestellabbrüche zu behandeln.

Über das Erzwingen oder Vermeiden von Transitionen können wir die gewünschte Anforderung an den Adapter jedoch nicht umsetzen. Die Bestellabbrüche zu verbieten bedeutet, dass ein Kunde in keinem Ablauf die Bestellung abbrechen darf. Damit schließen wir Kunden aus, die nicht jeden Kaufvorgang abschließen. Wenn solche Kunden immer mal wieder etwas kaufen, möchten wir sie aber nicht komplett ausschließen. Analog, wenn wir die Zahlungsabwicklung erzwingen, bedeutet das, dass die Zahlungsabwicklung in jedem Ablauf statt finden muss. Wir schließen damit die Kunden aus, die auch mal einen Bestellvorgang abbrechen. Unser Wunsch ist, dass der es wenigstens in einem Ablauf möglich ist, dass die Zahlungsabwicklung stattfindet.

Stahl und Wolf geben die zu überdeckenden Transitionen und Plätze als Menge an. Sie modifizieren die Bedienungsanleitung so, dass jedes Element dieser Menge überdeckt wird. Mit einer globalen Annotation über die Zustände der Bedienungsanleitung charakterisieren sie, welche Zustände in einem korrekten Partner enthalten sein müssen, damit die Überdeckung in der Interaktion erreicht wird.



(a) N_1 (b) E

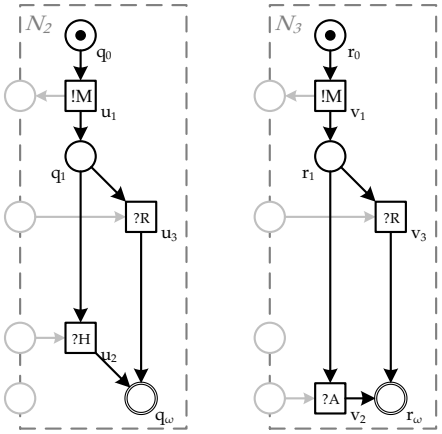
Abbildung 29: Getränkeautomat mit Engine

Die beiden vorgestellten Techniken zum Erzwingen und Vermeiden von Transitionen und zum Überdecken von Plätzen und Transitionen erweitern die Möglichkeiten, einen Adapter zu synthetisieren. Diese Techniken nutzen zwar Bedienungsanleitungen, basieren aber auf dem zugrunde liegenden allgemeinsten Kommunikationspartner. Somit können wir die Resultate direkt auf unsere Technik anwenden.

4.4.2 *Verhaltensoptimierung*

Mit den Techniken aus dem vorherigen Abschnitt können wir bereits die Anwesenheit und Abwesenheit von bestimmten Transitionen beeinflussen. Allerdings können wir nicht den Fall ausdrücken, dass wir auf eine Transition verzichten wollen, sofern das die Existenz eines Adapters nicht auszuschließt. Das Beispiel in den Abbildungen 29 und 30 veranschaulicht diesen Fall.

Wir greifen das Beispiel des Getränkeautomaten wieder auf und sehen in Abbildung 29a den Getränkeautomaten N_1, der nach Erhalt eines Betrages in Euro (?E) die Wahl für Cola (?c) oder Tee (?t) das entsprechende Getränk liefert (!C beziehungsweise !T). Er darf hier aber auch in der Anfangsmarkierung verbleiben.



(a) N_2 (b) N_3

Abbildung 30: Zwei verschiedene Kunden: N_2, der ein Heißgetränk möchte, und N_3, der ein alkoholisches Getränk möchte; beide können auch ihr Geld zurückerhalten

Die Engine E in Abbildung 29b implementiert wie zuvor die Regeln R_1: M → E (Münze in Eurobetrag) , R_2: → c (Auswahl Cola), R_3: → t (Auswahl Tee), R_4: C → S (Cola ist ein Softdrink) und R_5: T → H (Tee ist ein Heißgetränk). Zusätzlich führen wir eine Regel ein, die es uns erlaubt, eine Münze als Rückerstattung an den Kunden zurückzugeben: R_6: M → R. In der Schnittstelle der Engine gibt es auch einen Kanal A für alkoholische Getränke, den allerdings keine Regel nutzt.

In Abbildung 30 sehen wir zwei verschiedene Kunden. Der erste Kunde N_2 in Abbildung 30a sendet eine Münze (!M) und erwartet dann ein Heißgetränk (?H). Alternativ akzeptiert er auch die Rückerstattung seiner Münze (?R). Der zweite Kunde N_3 in Abbildung 30b verhält sich genauso wie N_2, er erwartet jedoch anstelle eines Heißgetränks ein alkoholisches Getränk (?A).

Für beide Kunden können wir mit der gegebenen Engine einen Adapter synthetisieren. Für den ersten Kunden N_2 bewirkt der Adapter, dass der Kunde nichtdeterministisch das gewünschte Heißgetränk oder die Rückerstattung erhält. Der zweite Kunde N_3 erhält ausschließlich die Rückerstattung.

Im zweiten Fall können wir es als positiv erachten, dass der Kunde die Rückerstattung bekommt und das Gesamtsystem terminiert, obwohl der Kunde sein Wunschgetränk

nicht bekommt. Im ersten Fall jedoch wäre es erstrebenswert, den Fall der Rückerstattung auszuschließen. Der Kunde möchte lieber das Getränk statt seines Geldes, und der Getränkeautomat soll Getränke verkaufen.

Wir möchten nun ausdrücken, dass die Rückerstattung nur dann Teil des Adapters ist, wenn dies für das korrekte Verhalten unverzichtbar ist. Wenn ein Kunde ein Getränk bekommen kann, wollen wir die Rückerstattung ausschließen, anderenfalls soll der Kunde wenigstens sein Geld zurückerhalten. Diese Spezifikation soll unabhängig vom Kunden sein. Mit den Mitteln aus dem vorherigen Abschnitt können wir dies jedoch nicht ausdrücken. Wenn wir erzwingen, dass die Münze als Eurobetrag weitergereicht wird (Regel R_1), oder vermeiden, dass sie rückerstattet wird (Regel R_6), dann können wir keinen Adapter für den zweiten Kunden N_3 synthetisieren. Auch mit Hilfe des Überdeckens von Transitionen kommen wir entweder zu den ursprünglichen Adaptern, oder wir finden keinen Adapter für N_3.

Wir betrachten deshalb eine Möglichkeit [35], den allgemeinsten Kommunikationspartner zu berechnen und anschließend Verhalten aus diesem zu entfernen. Als Mittel dient uns hier eine Zuordnung von Kosten zu Kommunikationslabeln und eine anschließende Optimierung dieser Kosten.

In Definition 24 haben wir $port_C$ als die Schnittstelle der Engine zum Controller definiert. Wir definieren nun eine *Kostenfunktion* $c : port_C \to \mathbb{N}$, die jedem Kanal in $port_C$ eine natürliche Zahl als deren Kosten zuweist. Im allgemeinsten Kommunikationspartner beschriften wir die einzelnen Kanten entsprechend ihres Labels mit Kosten.

Im so beschrifteten allgemeinsten Kommunikationspartner betrachten wir nun alle Pfade vom Anfangszustand zu einem Endzustand und summieren jeweils die Kosten entlang der Pfade auf. Wir streichen nun so Kanten und Zustände aus dem allgemeinsten Kommunikationspartner, sodass der resultierende Partner immer noch korrekt und die Summe der Kosten auf dem teuersten Pfad minimal ist. Minimal bedeutet, dass wir den teuersten Pfad nicht entfernen können, ohne die Korrektheit des Partners zu verletzen beziehungsweise, wenn wir statt diesem Pfad einen anderen drin gelassen hätten, einen Partner mit geringeren Kosten gefunden hätten. Wir streichen Kanten jedoch nur dann aus dem allgemeinsten Kommunikationspartner, wenn dadurch die Kosten des teuersten Pfades kleiner werden. Aller Pfade, deren Kosten geringer sind als die des teuersten Pfades, bleiben enthalten, um dem Adapter möglichste viel Verhalten zu erlauben.

Um zu entscheiden, ob wir immer noch einen korrekten Partner betrachten, können wir das Konzept der Bedienungsanleitung [61] nutzen. Jeder Zustand des allgemeinsten Kommunikationspartner wird mit einer Annotation über seine ausgehenden Kanten versehen. Wenn wir eine Kante streichen, müssen wir sicherstellen, dass die Annotation immer noch bezüglich der übrigen Kanten erfüllbar ist.

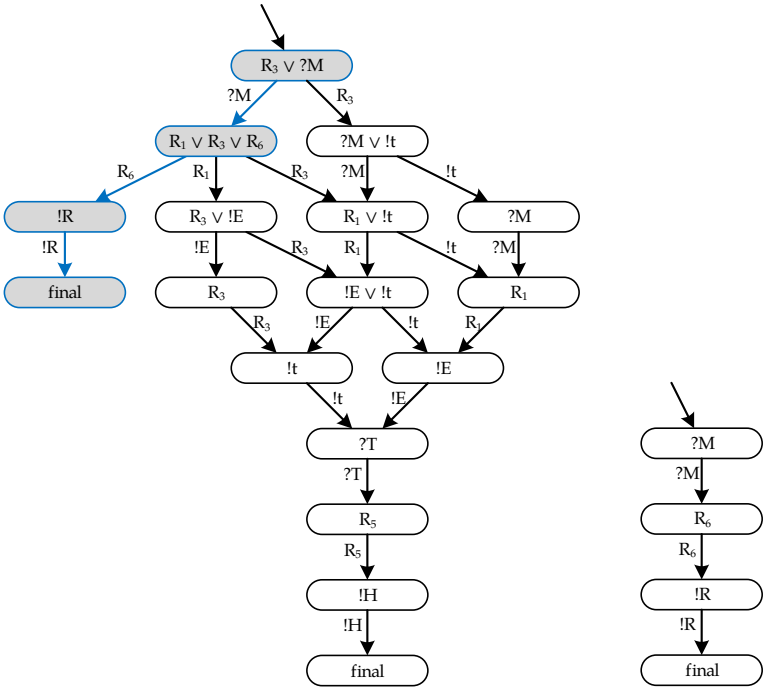
In Abbildung 31 sehen wir ein Beispiel für Bedienungsanleitungen für die beiden Kunden N_2 und N_3. Die Struktur entspricht dem allgemeinsten Kommunikationspartner, die Beschriftung an einer Kante bedeutet, dass sich der allgemeinste Kommunikationspartner mit der entsprechenden Transition in der Engine E synchronisiert. Jeder Zustand ist mit einer aussagenlogische Formel über die ausgehenden Kanten des Zustandes annotiert. Die Belegung eines Literals ist genau dann wahr, wenn die entsprechende Kante vorhanden ist. Das Literal final besagt, dass ein Kommunikationspartner in diesem Zustand terminieren darf.

Da wir auf die Rückerstattung wenn möglich verzichten wollen, weisen wir der Regel R_6: $M \rightarrow R$, beziehungsweise dessen Kommunikationslabel, positive Kosten zu, zum Beispiel 3, alle anderen Kommunikationslabel haben die Kosten 0.

Wir betrachten zuerst die Bedienungsanleitung für das System mit dem ersten Kunden N_2 in Abbildung 31a. Vom Anfangszustand mit der Annotation $R_3 \vee ?M$ ist der teuerste Pfad der, der immer links über die grauen Zustände führt. Zwischen den Zuständen $R_1 \vee R_3 \vee R_6$ und $!R$ finden wir das Label R_6, sodass die Kosten auf diesem Pfad in der Summe 3 ergeben. Alle anderen Pfade haben Kosten 0, denn jedes andere Label hat Kosten 0.

Die Kante R_6 entfernen wir nun, und mit ihr die nachfolgenden Zustände. Die Annotation $R_1 \vee R_3 \vee R_6$ ist erfüllt, weil es eine Disjunktion ist, und die Kanten mit R_1 und R_3 noch vorhanden sind. Durch Streichen der genannten Kante, erhalten wir somit einen Kommunikationspartner, der korrekt ist, aber die Regel R_6 nicht anwendet. Die Kosten in diesem Partner sind nun auf jedem Pfad 0, weil nur Label mit den Kosten 0 übrig bleiben, und somit minimal.

Für das zweite Beispiel für den zweiten Kunden N_3 sehen wir die Bedienungsanleitung in Abbildung 31b. Hier gibt es nur einen Pfad, der aufgrund des Labels R_6 in der Summe die Kosten 3 hat. Da jeder der ersten drei Zustände genau eine ausgehende Kante hat, die gleichzeitig Teil der entsprechenden Annotation ist, können wir keine Kante entfernen. Ansonsten wäre die entsprechende Annotation nicht mehr erfüllbar



(a) OG mit N_2 (b) OG mit N_3

Abbildung 31: Bedienungsanleitungen für das komponierte System aus Getränkeautomaten N_1, Engine E und dem ersten Kunden N_2 (links) beziehungsweise dem zweiten Kunden N_3 (rechts).

und somit kein korrektes Verhalten möglich. Die Regel R_6 ist daher unverzichtbar für korrektes Verhalten.

Wie wir sehen, können wir über eine Kostenminimierung auf dem allgemeinsten Kommunikationspartner beziehungsweise der Bedienungsanleitung unerwünschtes Verhalten ausschließen, wenn es nicht zwingend erforderlich für die Korrektheit eines Adapters ist. Über die Kosten für die einzelnen Kommunikationslabeln können wir sogar stufenweise priorisieren, welche Label und somit Regeln uns wichtiger sind. Damit können wir das Verhalten noch stärker einschränken.

Auf eine ausführliche Diskussion des Algorithmus zur Kostenoptimierung einer Bedienungsanleitung verzichten wir hier. Sie ist jedoch in einer früheren Publikation [35] beschrieben. Ein Nachteil dieses Algorithmus ist jedoch, dass er nur auf azyklischen Bedienungsanleitungen arbeitet. In der Arbeit von Sürmeli [103, 104] wird diese Beschränkung aufgehoben. Zudem lassen sich bei ihm beliebige Transitionen mit Kosten versehen und nicht nur welche, die Kommunikationslabel tragen. Dies ermöglicht die beschriebene Optimierungsidee auf beliebige Adapter und das Verhalten der gegebenen offenen Netze anzuwenden.

Was wir mit der Idee der Optimierung nicht erreichen, ist es einen Adapter zu generieren, der zur Laufzeit Verhalten priorisiert und immer das gewünschter Verhalten bevorzugt und unerwünschtes Verhalten nur wählt, wenn es nicht anders möglich ist. Insbesondere wenn wir für viele verschiedene Kunden immer wieder neu einen Adapter generieren müssen, erlaubt es uns der Optimierungsschritt, einen optimalen Adapter in einem Schritt zu finden. Gerade in dem Fall, wenn wir verschiedenen Priorisierungen des Verhaltens vornehmen, wäre der Ansatz sonst, zuerst nun mit den favorisierten Transformationsregeln zu versuchen, einen Adapter zu erzeugen. Wenn dies fehlschlägt, nehmen wir weitere Regeln hinzu und probieren es erneut, und so weiter. Erst im letzten Schritt mit allen Transformationsregeln erfahren wir dann, ob überhaupt ein Adapter existiert.

4.5 ZUSAMMENFASSUNG

In diesem Kapitel haben wir betrachtet, welche Eigenschaften die von uns im vorherigen Kapitel betrachtete Technik zur Adaptersynthese hat, und welche Möglichkeiten wir haben, die Technik anzuwenden und zu erweitern.

Wir haben festgestellt, dass wenn für die gegebenen offenen Netze ein Controller existiert, dass wir dann immer einen Adapter finden können, vorausgesetzt die Semantik der Transformationsregeln lässt dies zu. Existiert für eines der offenen Netze hingegen kein Controller, dann existiert auch kein Adapter. Das Thema führen wir in Kapitel 7 fort, in dem wir Empfehlungen für Transformationsregeln für den Fall geben, dass kein Controller synthetisiert werden kann.

Bei der Betrachtung der Schnittstelle zwischen Engine und Controller des Adapters hat sich ergeben, dass die synchrone Schnittstelle häufiger zu einem Controller führt. Im Falle beschränkten Verhaltens der Engine und Kommunikation und einer angemessenen Schranke, unterscheiden sich synchrone und asynchrone Schnittstelle diesbezüglich jedoch nicht.

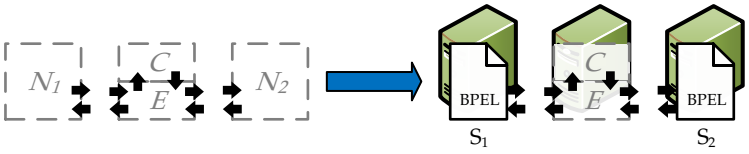
Wir haben gezeigt, dass sich das Konzept der Engine auf andere Szenarien wie zum Beispiel Firewallsysteme übertragen lässt. Zwar erfordert dies eine Anpassung der Semantik einer Engine, jedoch können wir mit einer solchen Engine Filterregeln einer Firewall ausdrücken. Der Controller ermöglicht solche Systeme automatisiert zu steuern.

Zuletzt haben wir ausgenutzt, dass der Controller des Adapters mit bereits vorhandenen Techniken berechnet werden kann. Somit stehen uns Erweiterungen dieser Technik ebenfalls zur Verfügung. So können wir sowohl Transitionen der Engine als auch der gegebenen offenen Netze erzwingen, vermeiden oder überdecken, und Verhalten aus dem Adapter entfernen, dass für einige offene Netze notwendig, für andere aber unerwünscht ist.

Insgesamt steht uns damit eine sehr flexible Technik zur Verfügung. Sie findet Anwendung für kommunizierende, offene System, wobei die Engine an die Kommunikationssemantik angepasst werden kann. Uns schränkt ausschließlich die Semantik von Nachrichten in der Anwendbarkeit der Technik ein. Mit der Controllersynthese generieren wir unabhängig davon den für unsere Anforderungen passenden Kontrollfluss.

5 PRAKTISCHE UMSETZUNG

Wie implementieren wir die Technik zur Adaptersynthese, sodass real existierende Prozesse transparent adaptiert werden können?



5.1 PROBLEMSTELLUNG

Als Motivation eine Technik zur Adaptersynthese einzuführen, dient zu großen Teilen, existierende und tatsächlich ausgeführte offene Systeme zu adaptieren. Wir wollen daher in diesem Kapitel vorstellen, wie wir kommunizierende Geschäftsprozesse, die in der Sprache WS-BPEL 2.0 [83] implementiert sind, adaptieren können.

Die *Web Services Business Process Execution Language* (kurz WS-BPEL oder BPEL) ist ein Standard zur Beschreibung von ausführbaren Geschäftsprozessen, die über das Internet kommunizieren. Die Sprache hat eine breite Unterstützung [5, 45, 84] erfahren und ist dem entsprechend häufig anzutreffen, wenn es um die Ausführung von Webservices geht.

Um einen Geschäftsprozess, der in BPEL angegeben ist, adaptieren zu können, müssen wir BPEL-Prozesse in offene Netze umwandeln, um unsere Technik anzuwenden. Das resultierende Adapter muss dann zur Ausführung gebracht werden.

Für WS-BPEL gibt es verschiedene Übersetzung in formale Sprachen, unter anderem von Lohmann in offene Netze [56]. Mit Hilfe dieser Semantik übersetzt das Werkzeug

BPEL2oWFN [59] einen BPEL-Prozess in ein offenes Netz unter Abstraktion von Daten. Trotz dieser Abstraktion hat sich die Semantik als sinnvoll erwiesen.

Eine Möglichkeit, den fertigen Adapter zur Ausführung zu bringen, wäre die Übersetzung des offenen Netzes in eine BPEL-Prozess. Allerdings erlaubt es die Struktur eines offenen Netzes nur selten, einen äquivalente Übersetzung in einen BPEL-Prozess zu finden [49].

Brogi und Popescu [15] erzeugen direkt einen Adapter als BPEL-Spezifikation, sodass dieser direkt ausgeführt werden kann. Allerdings ist der Ansatz auf bestimmte Patterns beschränkt.

Alternativ besteht die Möglichkeit, den Adapter als offenes Netz auszuführen und mit den gegeben BPEL-Prozessen kommunizieren zu lassen. Hier besteht die Herausforderung darin, ein Petrinetz nicht nur zu simulieren, sondern die tatsächlichen Transformationsregeln anzuwenden und mit den BPEL-Prozessen zu kommunizieren. In diesem Kapitel verfolgen wir genau diese Idee.

Wir stellen zwei Werkzeuge vor. MARLENE implementiert die Technik aus 3 zur Adaptersynthese. Für eine Menge von offenen Netzen und Transformationsregeln erzeugt MARLENE Engine und Controller und liefert diese als offene Netze zurück. Das Werkzeug CHARLOTTE nutzt die Ergebnisse von MARLENE und führt Engine und Controller so aus, dass wir BPEL-Prozesse adaptieren.

AUSGANGSSITUATION Gegeben seien zwei offene Systeme S_1 und S_2 als BPEL-Prozesse, die in offene Netze N_1 und N_2 übersetzt werden. Weiterhin sei eine Menge $\mathcal{R}$ von Transformationsregeln gegeben. Wir berechnen mit MARLENE einen Adapter, den wir mit CHARLOTTE ausführen, um S_1 und S_2 zu adaptieren.

GLIEDERUNG In diesem Kapitel betrachten wir zwei Werkzeuge: MARLENE zum Erzeugen eines Adapters und CHARLOTTE zum Ausführen eines Adapters. Wir gehen zuerst auf das Werkzeug MARLENE in Abschnitt 5.2 ein. Anschließend betrachten wir in Abschnitt 5.3, welche Maßnahmen ergriffen werden müssen, um einen Adapter als offenes Netz auszuführen, und wie das Werkzeug CHARLOTTE dies umsetzt. Wir fassen das Kapitel in Abschnitt 5.4 zusammen.

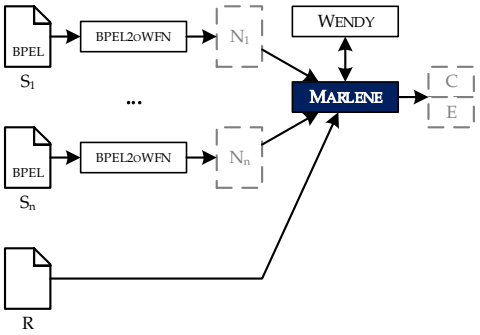


Abbildung 32: Das Werkzeug Marlene zum Erzeugen eines Adapters

5.2 Synthesetool Marlene

Die Aufgabe des Werkzeugs Marlene [36] ist es, aus einer gegeben Menge an offenen Netzen $N_1, \ldots, N_n$ und einer Menge $\mathcal{R}$ an Transformationsregeln Engine E und Controller C eines Adapters zu synthetisieren. Es setzt somit die Technik aus Kapitel 3 um.

Marlene reiht sich in eine Reihe von Werkzeugen ein, die für die Analyse, Verifikation und Synthese offener Systeme auf service-technology.org bereit gestellt werden. Die wesentlichen Merkmale von Marlene sind die plattformübergreifende Implementation in C++ , ein einheitliches Format zum Datenaustausch, insbesondere für offene Netze und Controller, und flexible Wahl der genutzten Werkzeuge für die einzelnen Schritte. Wie Marlene mit anderen Werkzeugen interagiert, um BPEL-Prozesse adaptieren zu können, sehen wir in Abbildung 32.

Die Eingabe für Marlene erzeugen wir aus gegebenen BPEL-Prozessen S_1 bis S_n. Das Werkzeug BPEL2oWFN übersetzt jeden einzelnen dieser Prozesse S_i in ein offenes Netz N_i. Die Transformationsregeln R sind in einem proprietären Textformat gegeben. Aus den offenen Netzen und den Transformationsregeln konstruiert Marlene eine Engine E. Für die Komposition aus Engine und den offenen Netzen ruft Marlene ein Werkzeug zur Controllersynthese, wie zum Beispiel Wendy, auf und erhält so einen Controller C, sofern ein solcher existiert.

Sowohl bei der Ein- als auch bei der Ausgabe erlaubt Marlene Flexibilität. Die Anzahl der gegeben offenen Netze muss mindestens 1 sein und kann beliebig groß

werden. Sollten die offenen Netze nicht paarweise disjunkt sein bezüglich ihrer Interfaces, kann MARLENE jedes Interface kanonisch umbenennen, sodass der Bezug zu den Transformationsregeln nicht verloren geht. Die Transformationsregeln werden in einer Datei angegeben. Das entsprechende Dateiformat ließe sich leicht erweitern, um zum Beispiel die Semantik einer Transformationsregeln direkt in der Datei anzugeben. Mit verschiedenen Parametern können wir MARLENE so steuern, dass ein gewünschtes Korrektheitskriterium vorausgesetzt wird, dass Engine oder Controller einzeln ausgegeben werden, oder dass der Controller möglichst klein ist.

Da MARLENE auf eine Bibliothek zur Manipulation von offenen Netzen zurückgreifen kann, gestaltete sich die Implementation recht einfach. Die gegebenen offenen Netze werden eingelesen genauso wie die Transformationsregeln. Anhand der Interfaces der offenen Netze wird das Interface der Engine erzeugt und anschließend die Transformationsregeln umgesetzt. Für die Komposition der offenen Netze mit der Engine können wir wieder auf eine Bibliotheksfunktion zurückgreifen. Lediglich das Beschränken der Plätze musste neu implementiert werden. Das Kompositum dient als Eingabe für das Werkzeug zur Controllersynthese, deren Ausgabe mit der Engine komponiert und ausgegeben wird.

Aufgrund des modularen Aufbaus, lassen sich auch die nachfolgenden Ergebnisse dieser Arbeit leicht in MARLENE integrieren. So kann MARLENE nach der Synthese des Adapters einen zusätzlichen Schritt aufrufen, um einen Adapter zu verteilen (Kapitel 6), oder es gibt Diagnoseinformationen aus, falls der Controller nicht generiert werden konnte (Kapitel 7).

5.3 AUSFÜHREN EINES SYNTHETISIERTEN ADAPTERS

Wenn wir das Ziel verfolgen, BPEL-Prozesse zu adaptieren, sollte das Ergebnis der Adaptersynthese mit den laufenden BPEL-Prozessen interagieren können. Um in der Welt von BPEL-Prozessen zu bleiben, wäre es vorteilhaft, wenn der resultierende Adapter selbst ein BPEL-Prozess ist. Unsere Technik basiert jedoch auf offenen Netzen und eine direkte Übersetzung eines offenen Netzes in einen BPEL-Prozess ist nur in wenigen Fällen möglich [49]. Die Alternative besteht darin, den Adapter als offenes Netz auszuführen und die Interaktion mit den BPEL-Prozessen zu ermöglichen.

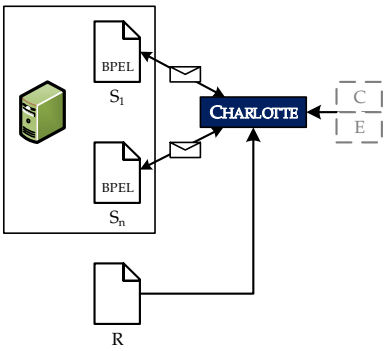


Abbildung 33: Das Werkzeug CHARLOTTE zum Ausführen eines Adapters

Für diese Aufgabe ist das Werkzeug CHARLOTTE [111] entstanden. Eine Übersicht über die Funktionsweise von CHARLOTTE bietet Abbildung 33.

Das Werkzeug CHARLOTTE ist eine Laufzeitumgebung für den synthetisierten Adapter und in der Lage mit ausgeführten BPEL-Prozessen zu kommunizieren. Die Kommunikation mit den BPEL-Prozessen S_1 bis S_n, die auf einem Server ausgeführt werden, realisiert CHARLOTTE durch die Nutzung entsprechender Kommunikationsprotokolle, die vom Server vorgegeben sind. Mit Hilfe der Engine E und dem Regelsatz R – insbesondere der konkreten Implementation für jede Transformationsregel – ist CHARLOTTE in der Lage, die Nachrichten der BPEL-Prozesse semantisch sinnvoll zu verarbeiten. Der Controller wird eigenständig entsprechend der Schaltregel ausgeführt und die Engine entsprechend gesteuert.

Wir sehen nachfolgend eine etwas detaillierten Blick auf die Aufgaben von CHARLOTTE.

5.3.1 *BPEL-Prozesse und deren Ausführung*

Ein BPEL-Prozess wird in einer speziellen Laufzeitumgebung ausgeführt, die den Prozess interpretiert und die Kommunikation realisiert. Im Rahmen von CHARLOTTE konzentrieren wir uns auf *Apache ODE* [5]. Die Anpassungen an weitere Umgebungen ist in der Regel ohne weiteres möglich.

Aus Sicht der Adaptersynthese ist der wichtigste Punkt einer Laufzeitumgebung die Art des Nachrichtenaustauschs. Das Interface eines BPEL-Prozesses wird in der *Web Service Description Language (WSDL)* [25] angegeben. Mit Hilfe einer WSDL-Beschreibung wissen wir genau, welche Nachrichtentypen ein BPEL-Prozess austauschen kann, und wie diese Nachrichten aufgebaut sind. Der Austausch erfolgt über das *Soap-Protokoll* [108]. Da CHARLOTTE in *C♯* implementiert ist, kann das Werkzeug auf entsprechende Bibliotheken zur Kommunikation über Soap und die Behandlung von WSDL-Nachrichtentypen zurückgreifen.

5.3.2 *Anwenden von Transformationsregeln*

Die Idee von CHARLOTTE ist, dass die Transformationsregeln direkt mit entsprechendem Code, zum Beispiel in JavaScript, angegeben werden. Um diese jedoch Anwenden zu können, müssen wir auf entsprechende Nachrichten zugreifen können.

Die Nachrichtentypen müssen Datentypen werden, Nachrichten dann entsprechende Objekte dieser Datentypen. Mit Hilfe eines abstrakten Datentyps wie einer Liste, Warteschlange oder Multimenge können wir einen einzelnen Platz repräsentieren. Während für einen Platz in einem offenen Netz nicht klar ist, welche Marke eine Transition von ihm konsumiert, können wir über geeignete Datenstrukturen die Semantik eines Platzes verändern. Solange das Konsumieren und Produzieren einer Marke in der Datenstruktur richtig umgesetzt ist, lässt sich dies mit der Petrinetzsemantik vereinbaren.

Beim Anwenden einer Transformationsregel müssen wir die benötigten Daten aus den entsprechenden Datenstrukturen auslesen, und das Ergebnis zurückschreiben. Jede einzelne Transformationsregel können wir so generisch als Funktion darstellen, die zuerst die benötigten Daten aus den Datenstrukturen ausliest, den Transformationscode des Nutzers ausführt und anschließend die erzeugten Daten in die Datenstrukturen speichert.

Wann welche Funktion und somit Transformationsregel angewendet wird, bestimmt der Controller.

5.3.3 *Simulation des Controllers*

Der Controller steuert, welche der Transformationsregeln angewendet werden. Da er als offenes Netz – also als Petrinetz – gegeben ist, müssen wir den Controller mit der entsprechenden Anfangsmarkierung gemäß der Schaltregel ausführen. Dazu müssen wir also die Netzstruktur intern repräsentieren, genauso Markierungen und den Effekt, den das Schalten einer Transition auslöst.

Wann immer eine Transition schaltet, die eine Nachricht mit der Engine austauscht, müssen wir sicher stellen, dass die zugehörige Funktion in der Implementation der Engine aufgerufen und Nachrichten transformiert werden.

Im Falle eines Konfliktes lassen wir über eine Zufallsgenerator entscheiden, welche Transition schaltet. Es sind auch andere Verfahren möglich, jedoch müssen diese sicherstellen, dass kein zyklisches Verhalten ausgelöst wird, dass keine Endmarkierung erreicht. Es könnte passieren, dass eine ganz bestimmte Transition schalten muss, damit wir eine Endmarkierung erreichen können. Wenn diese Transition jedoch im Konflikt mit einer anderen Transition steht, und diese zweite Transition immer bevorzugt schaltet, dann erhalten wir ein Verhalten, in der nie eine Endmarkierung erreicht wird. Somit würde die Implementation gegen die vom Controller vorgegeben Spezifikation verstoßen, weil Verhalten ausgeschlossen wird.

5.4 Zusammenfassung

Wir haben in diesem Kapitel zwei Werkzeuge kennengelernt. Das Werkzeug Marlene setzt die Technik zur Adaptersynthese aus Kapitel 3 um, und erzeugt aus gegebenen offenen Netzen und Transformationsregeln Engine und Controller. Außerdem implementiert Marlene die Ergebnisse aus Kapitel 6 zum Verteilen eines Adapters und aus Kapitel 7 zur Diagnose. Das zweite Werkzeug Charlotte nutzt die Ausgabe von Marlene um laufende BPEL-Prozesse zu adaptieren.

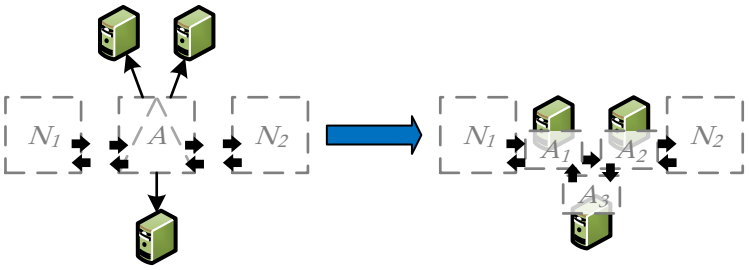
Die Konzepte des hier genutzten Ansatzes zur Adaptersynthese erlauben es uns, die Werkzeuge sehr modular aufzubauen und einfach zu erweitern. Die Trennung von Datenfluss und Kontrollfluss erlaubt es uns insbesondere, Engine und Controller getrennt auszuführen.

Teil II

ANWENDUNG DER TECHNIK

6 VERTEILEN

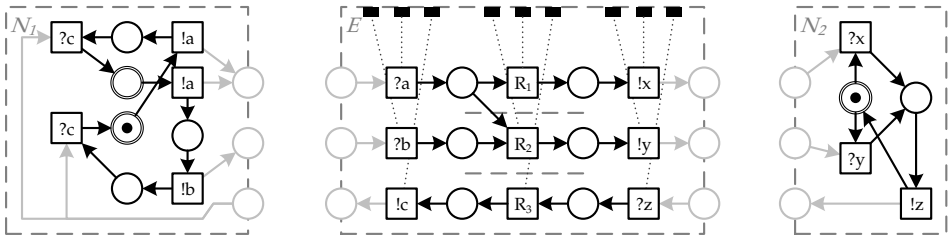
Wie können wir einen synthetisierten Adapter auf verschiedene Komponenten verteilen, wenn wir initial eine Verteilung der Transformationsregeln kennen?



6.1 PROBLEMSTELLUNG

Im vorherigen Kapitel haben wir gesehen, dass wir einen mit der vorgestellten Technik synthetisierten Adapter so ausführen können, dass dieser mit tatsächlich ausgeführten Geschäftsprozessen interagiert. Dabei läuft der Adapter auf einem einzigen System.

Da die Umsetzung einer Transformationsregel auch ein Geschäftsgeheimnis eines der beteiligten Parteien betreffen könnte, fragen wir uns, wie wir bestimmte Teile des Adapters ausgliedern können, sodass wir die Implementationen der einzelnen Transformationsregeln voneinander trennen. Weiterhin möchten wir zum Beispiel verteilte Hardwarekomponenten respektieren. Beim Getränkeautomaten aus den vorherigen Kapiteln, ist es vorstellbar, dass der Geldeinwurf, die Tasten zur Getränkewahl und die Getränkeausgabe jeweils auf unterschiedlichen Platinen realisiert werden. Konzeptuell sollen sich dies in einem Adapter wiederfinden.



(a) Netz N_1 (b) Engine E (c) Netz N_2

Abbildung 34: Eine zu verteilende Engine mit zwei offenen Netzen

6.1.1 *Beispiel*

Zur Veranschaulichung des Problems betrachten wir das technische Beispiel in Abbildung 34. Es zeigt zwei offene Netze N_1 und N_2 sowie eine Engine E. Wie durch die gestrichelten Linie in der Engine angedeutet, möchten wir die drei Transformationsregeln R_1, R_2 und R_3 auf einzelne Komponenten aufteilen, ohne dabei das Verhalten des Adapters zu verändern. Da dieses Beispiel für die Fragestellung relevante Aspekte enthält, die im Getränkeautomatenbeispiel nicht vorhanden sind, nutzen wir dieses Beispiel im Laufe des Kapitels. Im Adapter des Getränkeautomaten gibt es keine Konflikte, die aber kritisch für die Möglichkeit der Verteilung sind.

Das Netz N_1 in Abbildung 34a sendet zyklisch abwechselnd entweder die Nachricht a oder die Nachrichten a und b. Mit dem Empfang der Nachricht c wird das Senden jeweils quittiert und N_1 ist in einer Endmarkierung.

Das Netz N_2 in Abbildung 34c wartet in der Anfangsmarkierung auf die Nachricht x oder y. Wenn es eine dieser Nachrichten empfängt, sendet es die Nachricht z und geht zurück in die Anfangsmarkierung, die gleichzeitig eine Endmarkierung ist.

Die Engine E in Abbildung 34b implementiert drei Transformationsregeln: R_1: a $\rightarrow$ x, R_2: a, b $\rightarrow$ y und R_3: z $\rightarrow$ c. Die erste Regel benötigen wir, wenn N_1 nur ein a sendet, die zweite wenn N_1 sowohl ein a als auch ein b sendet. Die dritte Regel leitet die Quittung von N_2 an N_1 weiter.

Die drei Transformationsregeln sollen im Adapter auf unterschiedlichen Systemen ausgeführt werden. Für die dritte Transformationsregel R_3 ist das leicht ersichtlich, da sie andere Nachrichtentypen verarbeitet als R_1 und R_2. Im Gegensatz dazu ist

das bei den letztgenannten beiden Transformationsregeln nicht offensichtlich. Beide Transformationsregeln benötigen die Nachricht a, kommen aber nur abwechselnd zum Zug. Unabhängig davon, in welcher Komponente die Nachricht a liegt, müssen wir sicherstellen, dass sie für das Anwenden der Transformationsregel zur Verfügung steht.

Die zentrale Frage ist, wie wir einen Adapter in Komponenten aufteilen, und wie die Komponenten mit einander kommunizieren. Wir folgen in diesem Kapitel der Arbeit von Glabbeek, Goltz und Schicke [40] und somit auch deren Auffassung, dass Komponenten asynchron kommunizieren. Aus technischer Sicht ist dies ein sinnvolle Annahme, da zwischen dem Senden einer Nachricht und deren Empfang in Wirklichkeit immer Zeit vergeht. Synchroner Nachrichtenaustausch ist lediglich die Abstraktion eines entsprechenden Protokolls, dass auf asynchronem Nachrichtenaustausch beruht.

Für die Verteilung des Adapters auf Komponenten nutzen wir eine Verteilungsfunktion, die jedem Knoten des Adapters seine Komponente zuordnet. Wenn ein Platz und eine mit ihm verbundene Transition unterschiedlichen Komponenten zugeordnet sind, dann muss zwischen dem Zeitpunkt, in dem sich die Markierung des Platzes ändert, und dem Zeitpunkt, in dem die Transition schaltet, Zeit vergehen. In Wirklichkeit wird ein Komponente nicht tatsächlich synchron auf eine Ressource einer anderen Komponente zugreifen, sondern es wird immer einen zeitlichen Abstand geben.

Wir nutzen eine asynchrone Implementation, die genau diesen zeitlichen Abstand bewirkt bezüglich der Verteilung, und zerschneiden ein Netz entlang der dadurch eingeführten Kommunikationsplätze in einzelne Komponenten. Wir ändern durch diese Methode die Struktur des Netzes. Allerdings nutzen wir aus, dass im Fall der Step-Readiness-Äquivalenz van Glabbeek et al. eine spezielle Art von Konflikt identifiziert haben, den wir vermeiden müssen. Mit der gegeben Äquivalenz erhalten wir unter anderem Eigenschaften wie Verklemmungsfreiheit und Schwache Terminierung.

AUSGANGSSITUATION Gegeben sei ein synthetisierter Adapter und eine gewünschte Verteilung der Regeltransitionen auf Komponenten. Ziel ist es, den gesamten Adapter verteilen, sodass die Regeltransitionen den gewünschten Komponenten zugeordnet sind und die Komposition der Komponenten verhaltensäquivalent zum Adapter ist.

GLIEDERUNG Im Folgenden betrachten wir zuerst die Arbeit von Glabbeek, Goltz und Schicke [40] als Grundlage für das Verteilen von Adaptern. Abschnitt 6.2 führt ein, wie wir Komponenten asynchron verteilen, und unter welchen Voraussetzungen diese

Verteilung das Verhalten des ursprünglichen Netzes erhält. Anschließend beschreiben wir in Abschnitt 6.3, wie wir ausgehend von der Verteilung der Transformationsregeln den synthetisierten Adapter maximal verteilen, sofern dies möglich ist. Die Implementation der Idee geben wir in Abschnitt 6.4 an, bevor wir die Ergebnisse in Abschnitt 6.5 zusammenfassen.

6.2 VERTEILEN VON PETRINETZEN IN ASYNCHRONE KOMPONENTEN

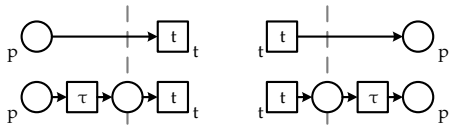
Der folgende Abschnitt basiert auf der Arbeit von van Glabbeek et al.[40] und führt Definitionen und Resultate zum Verteilen von gelabelten Petrinetzen ein. Wir betrachten eine Verteilungsfunktion für die Knoten. Bezüglich dieser wird das Netz asynchron implementiert. Van Glabbeek et al. identifizieren den verteilten Konflikt als Struktur, die Step-Readiness-Äquivalenz verletzt, die wir entsprechend einführen.

Die in diesem Abschnitt betrachteten Label bezeichnen beobachtbare *Aktionen*. In der Regel entspricht das Label dem Namen der Transition. Zusätzlich verwenden wir τ als Label, was bedeutet, dass das Schalten der entsprechenden Transition nicht beobachtet werden kann. Von den Kommunikationslabeln abstrahieren wir in diesem Abschnitt. Die Menge der beobachtbaren Aktionen bezeichnen wir mit Act.

Definition 34 (Gelabeltes Petrinetz)
Wir nennen $N = \langle P, T, F, \alpha, l \rangle$ ein *gelabeltes Petrinetz* genau dann, wenn $\langle P, T, F, \alpha \rangle$ ein Petrinetz und $l : T \to Act \dot{\cup} \{\tau\}$ eine Labelfunktion ist, die einer Transition entweder eine Aktion oder τ zuweist.

Van Glabbeek et al. schränken die Klasse der betrachteten Petrinetze auf 1-sichere, schlingenfreie Petrinetze ein. Beide Einschränkungen erleichtern, Schritte in einem Petrinetz, also das zeitgleiche Schalten mehrerer Transitionen, zu definieren. Die Einschränkungen bewirken außerdem, dass eine Transition nicht nebenläufig zu sich selbst schalten kann, und dass die Aktivierung einer Transition nur vom Vorbereich abhängt ohne auf den Nachbereich zu achten.

Das Resultat von van Glabbeek et al. lässt sich also direkt übertragen, wenn das Resultat der Adaptersynthese 1-sicher ist. Ausgangspunkt für uns ist dann ein 1-sicheres offenes Netz, für das wir eine kanonische Labelfunktion l annehmen, die jeder Transition ihren Namen als Aktion zuordnet.



(a) P-T-Schnitt (b) T-P-Schnitt

Abbildung 35: Verteilen eines Netzes, jeweils vorher (oben) und nachher (unten)

6.2.1 *Aufteilen von benachbarten Plätzen und Transition*

Wenn wir ein Netz verteilen, werden benachbarte Knoten des Netzes unterschiedlichen Komponenten zugeordnet. Aus praktischer Sicht sollte dann das Schalten der Transition und das Konsumieren beziehungsweise Produzieren einer Marke auf dem entsprechenden Platz unterschiedliche Ereignisse darstellen – es vergeht Zeit, um eine Ressource von einer Komponente zu einer anderen zu transportieren. Der Zugriff auf die Ressource findet nicht instantan statt. Abbildung 35 illustriert eine Möglichkeit, ein Netz *asynchron zu implementieren* und so das gewünschte Verhalten umzusetzen.

Links in Abbildung 35a sehen wir oben einen Platz p und eine Transition t, die, wie durch die gestrichelte Linie angedeutet, auf unterschiedliche Komponenten verteilt werden sollen. Da die beiden Knoten in unterschiedlichen Komponenten liegen sollen, nehmen wir an, dass erst eine Marke von p verschwindet, und danach t schaltet. Die Umsetzung sehen wir darunter: Eine τ-Transition konsumiert die Marke und produziert sie auf einem neuen Kommunikationsplatz. Von diesem kann t die Marke konsumieren und schalten.

In der rechten Abbildung 35b sehen wir den analogen Fall, wenn das Schalten einer Transition t vom Produzieren einer Marke auf Platz p getrennt wird. Dafür führen wir einen neuen Schnittstellenplatz und eine entsprechende τ-Transition ein.

Formal ordnen wir jedem Knoten, also jedem Platz und jeder Transition, eine Komponente zu. Wenn zwei durch die Flussrelation benachbarte Knoten in unterschiedliche Komponenten liegen, dann fügen wir eine τ-Transition und einen Schnittstellenplatz ein, die es uns erlauben, die einzelnen Komponenten als offene Netze zu beschreiben.

Definition 35 (Verteilungfunktions)
Eine *Verteilung* $\mathcal{D}$ eines gelabelten Petrinetzes N ordnet jedem Platz und jeder Transition eine *Komponente* zu $\mathcal{D} : (P \cup T) \to K$. Wir schreiben $x \equiv_{\mathcal{D}} y$, genau dann wenn $\mathcal{D}(x) = \mathcal{D}(y)$.

Wir interessieren uns nicht für die tatsächliche Komponente eines Knotens x. Wir betrachten daher nur, ob zwei Knoten x und y zur gleichen Komponente gehören ($x \equiv_{\mathcal{D}} y$) oder nicht ($x \not\equiv_{\mathcal{D}} y$). In der asynchronen Implementation eines Netzes fügen wir im zweiten Fall, wie in Abbildung 35 gezeigt, eine τ-Transition und einen Schnittstellenplatz ein.

Definition 36 (Implementation einer Verteilung)
Sei $N = \langle P, T, F, \alpha, l\rangle$ ein gelabeltes Petrinetz und $\mathcal{D}$ eine Verteilung. Dann ist die *asynchrone $\mathcal{D}$-Implementation* $I_{\mathcal{D}}(N) = \langle P \cup P^\tau, T \cup T^\tau, F', \alpha, l'\rangle$ mit

$$P^\tau = \{p_t \mid t \in T, p \in {}^\bullet t, p \not\equiv_{\mathcal{D}} t\} \cup \{p_t \mid t \in T, p \in t^\bullet, p \not\equiv_{\mathcal{D}} t\},$$

$$T^\tau = \{t_p \mid t \in T, p \in {}^\bullet t, p \not\equiv_{\mathcal{D}} t\} \cup \{t_p \mid t \in T, p \in t^\bullet, p \not\equiv_{\mathcal{D}} t\},$$

$$\begin{aligned} F' = \quad & \{(t,p) \mid t \in T, p \in t^\bullet, p \equiv_{\mathcal{D}} t\} \\ & \cup \{(p,t) \mid t \in T, p \in {}^\bullet t, p \equiv_{\mathcal{D}} t\} \\ & \cup \{(p,t_p),(t_p,p_t),(p_t,t) \mid t \in T, p \in {}^\bullet t, p \not\equiv_{\mathcal{D}} t\} \\ & \cup \{(t,p_t),(p_t,t_p),(t_p,p) \mid t \in T, p \in t^\bullet, p \not\equiv_{\mathcal{D}} t\} \end{aligned}$$

$$l' = \begin{cases} l(t) & \text{falls } t \in T \\ \tau & \text{falls } t \in T^\tau \end{cases}$$

BEMERKUNG: Bei van Glabbeek et al. wird nur für Kanten von einem Platz zu einer Transition eine τ-Transition eingeführt. Im umgekehrten Fall sehen die Autoren keine Notwendigkeit, da sie das Produzieren einer Marke als grundsätzlich asynchron betrachten, d. h., der Zeitpunkt, wann eine Marke tatsächlich auf dem Platz liegt, ist vernachlässigbar. Damit wir jedoch ein Netz sinnvoll in einzelne Komponenten verteilen können, wobei jede einzelne Komponente ein offenes Netz ist, fügen wir auch für Kanten von einer Transition zu einem Platz eine τ-Transition ein.

Das Zerschneiden der asynchronen Implementation eines Netzes in seine Komponenten geschieht entlang der neu eingefügten Schnittstellenplätze P^τ. Jeder Platz in P^τ hat genau eine Vor- und eine Nachtransition, sodass er genau der unidirektionalen Kommunikation entspricht, die wir für offene Netze fordern, und hat damit die Eigenschaften eines Schnittstellenplatzes, der bei der Komposition (Definition 11) von zwei offenen Netzen entsteht. Im Sinne von Lehmann [54] sind die Plätze in P^τ genau die Plätze zwischen Komponenten, und aufgrund ihres jeweils einelementigen Vor- und Nachbereichs bilateral. Damit lässt sich $I_{\mathcal{D}}(N)$ verhaltensäquivalent in Komponenten in Form von offenen Netzen zerschneiden.

Definition 37 (Zerschneiden in Komponentennetze)
Sei $N = \langle P, T, F, \alpha, l\rangle$, sei Ω die Menge der Endmarkierungen des zugrunde liegenden offenen Netzes, und sei $I_{\mathcal{D}}(N) = \langle P \cup P^\tau, T \cup T^\tau, F', \alpha, l'\rangle$ die asynchrone Implementation von N. Dann sei $N_K = \langle P_K, T_K, F_K, \alpha_K, \Omega_K, \mathcal{I}_K, \lambda_K\rangle$ ein *Komponentennetz* mit

$$
\begin{aligned}
P_K &= \{p \in P \mid \mathcal{D}(p) = K\},\\
T_K &= \quad \{t \in T \mid \mathcal{D}(t) = K\}\\
&\quad \cup \{t \in T^\tau \mid (p,t) \in F', \mathcal{D}(p) = K\}\\
&\quad \cup \{t \in T^\tau \mid (t,p) \in F', \mathcal{D}(p) = K\},\\
F_K &= F' \cap ((P_K \times T_K) \cup (T_K \times P_K)),\\
\alpha_K &= \alpha_{|P_K},\\
\Omega_K &= \{\omega_{|P_K} \mid \omega \in \Omega\},\\
\mathcal{I}_K &= \quad \{port_{K'} \mid \exists p \in P_K : (p,t) \in F, \mathcal{D}(t) = K' \neq K\}\\
&\quad \cup \{port_{K'} \mid \exists t \in T_K : (t,p) \in F, \mathcal{D}(p) = K' \neq K\}\\
&\quad \text{mit } port_{K'} = \{K'.!p_t \mid \exists t \in T_K : (t,p) \in F, \mathcal{D}(p) = K'\}\\
&\qquad\qquad \cup \{K'.?p_t \mid \exists t \in T_K : (p,t) \in F, \mathcal{D}(p) = K'\}
\end{aligned}
$$

$$
\lambda_K(port_X, t) = \begin{cases}
K'.!p_t & \text{falls } X = K',\ t \in T_K,\ (t,p_t) \in F',\\
& \mathcal{D}(p) = K'\\
K'.?p_t & \text{falls } X = K',\ t \in T_K,\ (p_t,t) \in F',\\
& \mathcal{D}(p) = K'\\
\lambda(port_X, t) & \text{sonst für } t \in T_K
\end{cases}
$$

Beim Verteilen müssen wir jedoch verhindern, dass wir das Verhalten des entstehenden Netzes soweit verändert, dass sich die Gültigkeit des Korrektheitskriterium ändert. Wenn das ursprüngliche Netz schwach terminiert, dann muss auch die Komposition der entstandenen Komponenten schwach terminieren und umgekehrt.

Dazu müssen wir zwei Punkte betrachten: Die Komposition der einzelne Komponenten muss wieder die Implementation von N ergeben, und die Implementation an sich, muss *verhaltensäquivalent* zu N sein. Der erste Punkt ergibt sich mit folgendem Lemma, den zweite Punkt betrachten wir im nächsten Abschnitt.

Lemma 38 (Komposition der Komponenten)
Für ein Netz N, dessen asynchroner Implementation $I_{\mathcal{D}}(N)$ und deren Komponentennetze $N_{K1}, \ldots, N_{Kn}$ gilt:

$$I_{\mathcal{D}}(N) = \bigoplus_{1 \leq i \leq n} N_{Ki}$$

(bis auf die Endmarkierungen).

Beweis.
Die Aussage ergibt sich unmittelbar aus Definition 37. Wir zerschneiden $I_{\mathcal{D}}(N)$ entlang der in Definition 36 eingeführten Schnittstellenplätze p_t. Ein solcher Platz p_t wird zu einem Kanal zwischen den beiden betroffenen Komponenten und die angrenzenden Transitionen erhalten jeweils ein Label, um entweder auf dem Kanal zu senden oder zu empfangen. Der Schnittstellenplatz wird nicht in die Komponentennetze übernommen, entsteht aber durch die entsprechend definierte Labelfunktion bei der Komposition.

Die Anfangsmarkierung wird auf die Plätze der Komponentennetze projiziert. Die Schnittstellenplätze entstehen erst bei der Implementation und sind daher leer. Jedes einzelne Komponentennetz hat dann genau eine Anfangsmarkierung, deren aller Kreuzprodukt in der Komposition wieder genau in der Anfangsmarkierung von $I_{\mathcal{D}}(N)$ resultiert.

Mit den Endmarkierungen verhält es sich im Prinzip analog. Allerdings ist nach der Projektion unklar, welche Endmarkierungen in Komponente i mit welcher Endmarkie-

rungen in Komponente j übereinstimmen. Bei der Komposition der n Komponenten entsteht somit eine Obermenge der Endmarkierungen von $I_{\mathcal{D}}(N)$. ∎

BEMERKUNG Dass wir beim Komponieren der einzelnen Komponentennetze eine Obermenge der ursprünglichen Endmarkierungen erhalten, ist in dem betrachten Fall der Adaptersynthese vernachlässigbar. Die Engine hat genau eine Endmarkierung, die sich auch in den Komponenten widerspiegelt. Das heißt, dass in jeder Endmarkierung in der Obermenge, die Engine genau diese eine Markierung haben muss. Der Controller könnte durch das Verteilen und anschließende Komponieren unter Umständen in mehr Markierungen terminieren, was aber nichts an der Korrektheit ändert.

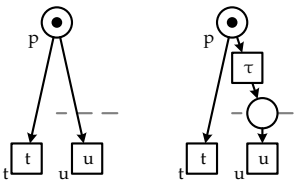
6.2.2 *Step-Readiness-Äquivalenz*

Bis hierher haben wir gesehen, wie wir ein gelabeltes Petrinetz N in Komponentennetze verteilen, sodass die Implementation $I_{\mathcal{D}}(N)$ durch die Komponenten korrekt dargestellt wird. Jetzt klären wir, wann sich ein Netz und dessen Implementation *äquivalent* verhalten. Insbesondere in Hinblick auf das Korrektheitskriterium, das wir während der Controllersynthese betrachten, führt nicht jede Verteilung $\mathcal{D}$ zu einer verhaltensäquivalenten Implementation.

Wir folgen weiterhin Glabbeek, Goltz und Schicke [40], und betrachten den Fall, dass ein gelabeltes Petrinetz N und dessen asynchrone Implementation $I_{\mathcal{D}}(N)$ *step-readiness-äquivalent* sind. Diese Äquivalenz betrachtet, welche Schritte nach einer Sequenz von Transitionen möglich sind. Sie hat den Vorteil, dass sie Verzweigungszeitpunkte, Kausalität und Divergenz in Petrinetzen bewahrt, und somit geeignet ist, eine große Klasse an Sicherheits- und Lebendigkeitseigenschaften zu bewahren. Sie ist dabei aber eine der schwächsten Äquivalenzen, die diese drei Aspekte bewahrt, weshalb in vielen Fällen die Implementation eines Netzes step-readiness-äquivalent zum Netz ist.

Abbildung 36 zeigt ein Problem für das äquivalente Verteilen eines Netzes N. Wir sehen links in Abbildung 36a einen markierten Platz p und zwei Transitionen t und u, die aufgrund von p im Konflikt stehen. In dieser Markierung, können wir entweder beobachten, wie die Transition t oder wie die Transition u schaltet.

Wir nehmen an, dass der Platz p und die Transition u in unterschiedliche Komponenten verteilt werden sollen. Eine Implementation dieser Verteilung gemäß Definition 36



(a) Konflikt (b) Verteilter Konflikt

Abbildung 36: Verteilter Konflikt

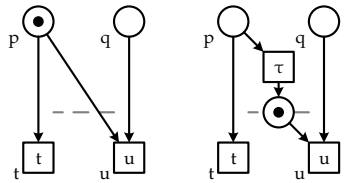
sehen wir in Abbildung 36b. Dass wir nun das Schalten von t beobachten können, ist nicht sicher. Da wir das Schalten der τ-Transition nicht beobachten können, ist es möglich, dass die τ-Transition schon geschaltet und somit t deaktiviert hat. Die Situation, in der zwei Transitionen im Konflikt stehen und unterschiedlichen Komponenten zugeordnet sind, nennen wir *verteilten Konflikt*.

Definition 39 (Verteilter Konflikt)
Sei $N = \langle P, T, F, \alpha, l \rangle$ ein gelabeltes Petrinetz und $\mathcal{D}$ eine Verteilung von N. N hat genau dann einen *verteilten Konflikt*, wenn

- es zwei Transitionen $t, u \in T$ mit $t \neq u$ gibt,
- die einen gemeinsamen Vorplatz $p \in ({}^\bullet t \cap {}^\bullet u)$ haben,
- der eine anderen Komponente als u zugeordnet ist ($p \not\equiv_{\mathcal{D}} u$), und
- ein Markierung μ erreichbar ist ($\alpha \Rightarrow \mu$), die t aktiviert (${}^\bullet t \subseteq \mu$).

Abbildung 37 verdeutlicht, dass ein verteilter Konflikt problematisch ist. Wenn eine Transition u noch von weiteren Plätzen als p abhängt und nicht aktiviert ist.

In Abbildung 37a beobachten wir genau das Schalten der Transition t. Die Transition u kann nicht schalten, weil der unmarkierte Platz q dies verhindert. Verteilen wir den Platz p und die Transition u auf unterschiedliche Komponenten, kann es, wie in Abbildung 37b zu sehen, passieren, dass die τ-Transition schaltet und danach keine



(a) N (b) $I_{\mathcal{D}}(N)$

Abbildung 37: Problematischer verteilter Konflikt

Transition mehr aktiviert ist. Wir beobachten also im ersten Fall das Schalten von t, im zweiten Fall jedoch nicht. Die beiden Netze sind nicht step-readiness-äquivalent.

Die *Step-Readiness-Äquivalenz* sagt aus, dass zwei Netze genau dann äquivalent sind, wenn sie nach einem beliebigen endlichen Ablauf, der in beiden Netzen möglich ist, genau die gleichen *Schritte* schalten können. Ein Schritt beschreibt dabei das zeitgleiche Schalten mehrerer Transitionen. Die folgende Definition respektiert dabei, dass ein Netz nach Schalten eines Schrittes immer noch 1-sicher ist.

Definition 40 (Schritt (Petrinetz))
Sei N ein Petrinetz. Sei S eine nichtleere Menge von Transitionen ($\emptyset \subsetneq S \subseteq T$). Wir nennen S genau dann einen *Schritt* von μ nach μ', wenn

- alle Transitionen in S aktiviert sind, d. h., dass für alle $t \in S$ gilt, dass ${}^\bullet t \subseteq \mu$ und $(\mu \setminus {}^\bullet t) \cap t^\bullet = \emptyset$,
- die Transitionen in S unabhängig sind, d. h., dass sie nicht miteinander im Konflikt stehen, also für alle $t, u \in T$ mit $t \neq u$ gilt, dass ${}^\bullet t \cap {}^\bullet u = \emptyset$ und $t^\bullet \cap u^\bullet = \emptyset$, und
- μ' eine Nachfolgemarkierung von μ ist, in der von allen Vorplätzen der Transitionen in S die Marken entsprechend entfernt und auf den Nachplätzen produziert werden, also $\mu' = (\mu \setminus {}^\bullet S) \cup S^\bullet$.

In Kapitel 2 haben wir nur das Schalten einzelner Transitionen eingeführt. Dieser Fall wird von Schritten überdeckt, da ein Schritt insbesondere eine einelementige Menge

sein kann – also eine einzelne Transition. Damit wissen wir im folgenden, dass, wann immer alle Schritte bewahrt werden, dann auch das Schalten einzelner Transitionen bewahrt wird.

Zwei gelabelte Petrinetze sind dann äquivalent, wenn sie immer die gleichen Schritte schalten können. Das heißt, wann immer wir die gleiche Sequenz an beobachtbaren Aktionen in beiden Netzen gesehen haben, dann müssen in beiden Netzen die gleichen Schritte möglich sein. Insbesondere darf das Schalten einer τ-Transition niemals einen Schritt verhindern.

Definition 41 (Step-Readiness-Äquivalenz)
Sei $N = \langle P, T, F, \alpha, l \rangle$ ein gelabeltes Petrinetz, $\vec{\sigma} \in Act^*$ eine Sequenz und $S \subseteq \mathbb{N}^{Act}$ eine Menge von Multimengen über den Aktionen. Wir nennen $\langle \vec{\sigma}, S \rangle$ ein *Step-Ready-Paar* von N genau dann, wenn eine Markierung μ existiert, die von der Anfangsmarkierung α über $\vec{\sigma}$ erreichbar ist ($\alpha \overset{\vec{\sigma}}{\Longrightarrow} \mu$), keine τ-Transition aktiviert ist ($\mu \overset{\tau}{\nrightarrow}$), und $S = \{A \in \mathbb{N}^{Act} | \mu \overset{A}{\longrightarrow}\}$ umfasst ausschließlich Schritte als Multimengen aktivierter Aktionen.

Mit $\mathfrak{R}(N)$ bezeichnen wir die Menge aller Step-Ready-Paare von N. Zwei Netze N und N' sind genau dann *step-readiness-äquivalent* ($N \equiv_{\mathfrak{R}} N'$), wenn $\mathfrak{R}(N) = \mathfrak{R}(N')$.

Van Glabbeek et al. zeigen, dass man ein Netz mit verteiltem Konflikt nicht step-readiness-äquivalent asynchron implementieren kann. Solange wir beim Verteilen keinen verteilten Konflikt einführen, können wir ein Netz beliebig verteilen und asynchron implementieren. Dies nutzen wir im folgenden Abschnitt aus, um ausgehend von den Transformationsregeln den gesamten Adapter zu verteilen, sofern dies möglich ist.

Lemma 42 (Step-Readiness-Äquivalenz und verteilter Konflikt)
Sei $N = \langle P, T, F, \alpha, l \rangle$ ein gelabeltes Petrinetz und $\mathcal{D}$ eine Verteilung von N. Dann ist die asynchrone Implementation $I_{\mathcal{D}}(N)$ von N genau dann step-readiness-äquivalent zu N, wenn N keinen verteilten Konflikt enthält.

6.2.3 *Aktivierter Konflikt*

In unserem Beispiel (Abbildung 34) gibt es einen verteilten Konflikt zwischen den Transitionen R_1 und R_2, egal welcher Komponente wir den Konfliktplatz zuordnen. Daher ist es uns mit der gegeben Technik zur Implementation unmöglich, die beiden Transitionen auf unterschiedliche Komponenten zu verteilen. Van Glabbeek et al. stellen jedoch noch eine alternative Implementation vor, falls zwei Transitionen zwar in einem verteilten Konflikt, aber niemals tatsächlich im Konflikt stehen – also gleichzeitig aktiviert sind. Das heißt, in jeder erreichbaren Markierung ist maximal eine der beiden Transitionen aktiviert. Für den Fall, dass eine Markierung erreichbar ist, in der tatsächlich zwei im Konflikt stehende Transitionen aktiviert sind, befinden sich diese Transitionen in einer *Aktivierten-Konflikt-Relation.*

Definition 43 (Aktivierte-Konflikt-Relation)
Sei $N = \langle P, T, F, \alpha, l \rangle$ ein gelabeltes Petrinetz. Wir definieren die *Aktivierte-Konflikt-Relation* $\sim_{\text{AKR}} \subseteq T \times T$ als $t \sim_{\text{AKR}} u$ genau dann, wenn eine erreichbare Markierung μ existiert, sodass $\mu \xrightarrow{\{t\}}$ und $\mu \xrightarrow{\{u\}}$, aber $\mu \not\xrightarrow{\{t,u\}}$.

Wenn wir uns das initiale Beispiel genauer betrachten, fällt auf, dass die beiden Transitionen R_1 und R_2 zwar in einem verteilten Konflikt stehen, aber nicht in Aktivierter-Konflikt-Relation. Der nicht gezeigte Controller aktiviert diese beiden Transitionen immer nur abwechselnd und exklusiv, je nachdem, ob das offene Netz A nur eine Nachricht a oder beide Nachrichten a und b sendet.

In diesen Fall ist die Idee, die beiden im verteilten Konflikt, aber nicht in Aktiver-Konflikt-Relation stehenden Transitionen auf ihre Komponenten zu verteilen und jeder Transition erstmal eine eigene Kopie der Marke auf dem Konfliktplatz zukommen zu lassen. Da nur eine Transition tatsächlich schaltet, fügen wir weitere, nicht beobachtbare Aufräumtransitionen mit τ-Label ein, die die nicht mehr notwendige Marke entfernen. Folgende Implementation schlagen van Glabbeek, Goltz und Schicke vor.

Definition 44 (GGS-Implementation)
Sei $N = \langle P, T, F, \alpha, l \rangle$ ein gelabeltes Petrinetz. Mit $[t] = \{u \in T \mid t \sim^*_{\text{AKR}} u\}$ bezeichnen wir die Äquivalenzklasse aller mit t reflexiv und transitiv in Aktivierter-Konflikt-

Relation stehender Transitionen. Die GGS-Implementation von N ist definiert als $N' = \langle P \cup P^\tau, T \cup T^\tau, F', \alpha, l \rangle$ mit

$$
\begin{aligned}
P^\tau = \quad & \{p^{[t]} \mid p \in P; t \in p^\bullet\} \cup \{\textcircled{t} \mid t \in T\} \\
& \cup \{p_t^{[u]}, \overline{p}_t^{[u]} \mid p \in P; t, u \in p^\bullet; [t] \neq [u]\},
\end{aligned}
$$

$$
\begin{aligned}
T^\tau = \quad & \{\boxed{p} \mid p \in P\} \cup \{t' \mid t \in T\} \\
& \cup \{t_p^{[u]} \mid p \in P; t, u \in p^\bullet; [t] \neq [u]\},
\end{aligned}
$$

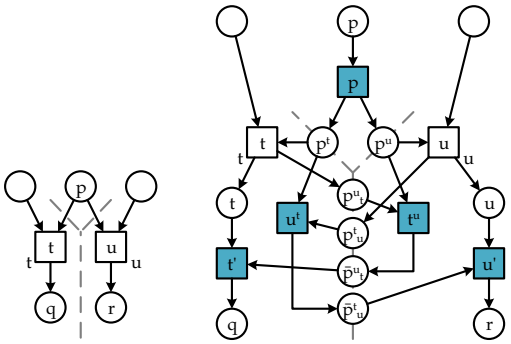
$$
\begin{aligned}
F' = \quad & \{(p, \boxed{p} \mid p \in P\} \\
& \cup \{(\boxed{p}, p^{[t]}), (p^{[t]}, t) \mid p \in P; t \in p^\bullet\} \\
& \cup \{(t, \textcircled{t}), (\textcircled{t}, t') \mid t \in T\} \\
& \cup \{(t', p) \mid t \in T; p \in t^\bullet\} \\
& \cup \{(t, p_t^{[u]}), (p_t^{[u]}, t_p^{[u]}), (t_p^{[u]}, \overline{p}_t^{[u]}), (\overline{p}_t^{[u]}, t'), (p^{[u]}, t_p^{[u]}) \\
& \qquad \mid p \in P; t, u \in p^\bullet; [t] \neq [u]\}
\end{aligned}
$$

$$
l'(t) = \begin{cases} l(t) & \text{falls } t \in T \\ \tau & \text{sonst, also } t \in T^\tau \end{cases}
$$

Wie diese Definition konkret an einem Beispiel aussieht, können wir in Abbildung 38 sehen. Der Platz p und die beiden im verteilten Konflikt stehenden Transitionen t und u werden auf jeweils eigene Komponenten verteilt. Wir nehmen gemäß Definition 44 an, dass t und u in keiner erreichbaren Markierung gleichzeitig aktiviert sind.

Die grau gestrichelte Linie dient der Unterscheidung der Komponenten. In der oberen Komponente sehen wir den Platz p. Eine gleichnamige Transition hat die Aufgaben, eine Marke auf p für jede der unteren Komponente einmal zu kopieren; es werden Marken auf den Plätzen p^t und p^u produziert. Der Aufbau der beiden unteren Komponenten ist symmetrisch.

Betrachten wir den Fall, dass die Transition t aktiviert ist. Damit ist u nach Voraussetzung nicht aktiviert. Da der Platz p^t markiert ist, kann die Transition t schalten. Wir müssen nun noch die Kopie, die wir der Komponente von u zur Verfügung gestellt haben, aufräumen. Dafür markieren wir beim Schalten von t den Platz p_t^u, der die Transition t^u aktiviert, welche die Marke auf p^u entfernt. Dass das Aufräumen fertig ist, wird



(a) N (b) N'

Abbildung 38: Implementation eines verteilten Konflikts

dann durch eine entsprechende Marke auf Platz $\overline{p}_t^u$ angezeigt. Abschließend schaltet die Transition t′ und markiert den ursprünglichen Nachbereich von t. In der Komponente von u wurden genau die Marken wieder entfernt, die durch diese Implementation entstanden sind.

Die farbig gefüllten Transitionen in der Abbildung sind genau die neu eingeführten Transitionen τ-Transitionen aus T^τ. Beobachtbar sind somit weiterhin nur die Transitionen t und u.

Neben der einfachen Implementation, bei der wir nur eine τ-Transition und einen Schnittstellenplatz einfügen müssen, um ein Netz verteilen zu können, steht uns mit der zweiten Implementation nun noch eine Alternative zur Verfügung. Im Folgenden nutzen wir die einfache Implementation, wenn die Verteilung keinen verteilten Konflikt enthält, und die komplexere Implementation, falls es zwar einen verteilten Konflikt zwischen zwei Transitionen gibt, aber diese nicht in Aktivierter-Konflikt-Relation stehen.

6.3 Vervollständigen der Verteilungsinformationen

Die Ausgangssituation ist, dass wir die Implementation bestimmter Transformationsregeln auf verschiedene Komponenten verteilen wollen. Wir nehmen an, dass wir für die Transitionen, welche die Transformationsregeln modellieren, die Verteilung

$\mathcal{D}(\mathrm{R}_1), \ldots, \mathcal{D}(\mathrm{R}_\mathrm{n})$ kennen. Für die restlichen Knoten des Adapters müssen wir noch eine Verteilung festlegen, sodass die Implementation dieser Verteilung verhaltensäquivalent zum synthetisierten Adapter ist.

Aus dem vorherigen Abschnitt wissen wir, welche Bedingungen wir an eine Verteilung stellen müssen, damit die Implementation step-readiness-äquivalent zum gegebenen Netz ist. Wir haben diesbezüglich noch die Freiheit über den Grad an Verteilung, den wir erreichen wollen. Entweder möchten wir möglichst wenige Komponenten haben, die dafür möglichst groß sind, oder wir wollen den Adapter maximal verteilen, d. h., die Komponenten sind minimal und können nicht feiner verteilt werden. Zwischen diesen beiden Extremen sind auch noch weitere Verteilungen möglich.

Wir betrachten eine *maximale Verteilung*, d. h., zwei unterschiedliche Knoten werden auf unterschiedliche Komponenten verteilt, wann immer das möglich ist. Diese Verteilung hat den Vorteil, dass sie eindeutig ist und sich jede andere, gröbere Verteilung durch Vereinigung verschiedener Komponenten ergibt.

6.3.1 *Algorithmus zum Verteilen eines Adapters*

Die Eingabe für unseren Algorithmus sind der synthetisierte Adapter und die Verteilung der Transformationsregeln $\mathcal{D}(R_1), \ldots, \mathcal{D}(R_n)$. Alle anderen Transitionen und Plätze kennzeichnen wir mit dem Symbol $\perp$, falls sie noch keiner Komponente zugewiesen wurden. Um auszudrücken, dass ein Knoten zu einer neuen, bis dahin nicht existierenden Komponente gehört, benutzen wir das Symbol ν. Wir bestimmen alle kritischen Strukturen, die wir jeweils auf eine Komponenten verteilen müssen, jeden anderen Knoten verteilen wir je eigene Komponente.

Wir dürfen Knoten nicht verteilen, wenn sie in Aktivierte-Konflikt-Relation gemäß Definition 44 stehen. Für eine Transition t gibt $[t]$ den *Konfliktcluster* aller mit t transitiv und reflexiv in Aktivierter-Konflikt-Relation stehenden Transitionen vor. Das bedeutet, dass wir diese Transitionen nicht auf verschiedene Komponenten verteilen können.

Ein Cluster kann der vorgegebenen Verteilung der Transformationsregeln widersprechen, wenn zwei Regeltransitionen in einem Cluster auf unterschiedliche Komponenten verteilt werden sollen. Dann lässt sich der Adapter nicht verteilen. Ansonsten können wir für die restlichen Knoten jeweils eine beliebige neue Komponente vergeben. Bei der Implementation müssen wir zusätzlich noch unterscheiden, ob wir die einfach

Implementation mit τ-Transition gemäß Definition 36 benutzen können, oder ob ein verteilter Konflikt vorliegt, sodass wir die Implementation aus Definition 44 benutzen müssen.

Wir definieren den *AKR-Cluster* als die Menge der Knoten, die aufgrund der Aktivierten-Konflikt-Relation auf die gleiche Komponente verteilt werden müssen. Dies umfasst nicht nur die in Aktivierter-Konflikt-Relation stehenden Transitionen, sondern auch deren Konfliktplätze im Vorbereich.

Definition 45 (AKR-Cluster)
Sei $N = \langle P, T, F, \alpha, l \rangle$ ein gelabeltes Petrinetz. Der *AKR-Cluster* einer Transition $t \in T$ ist definiert als $Clust(t) = \{u \mid u \in [t]\} \cup \{p \mid \exists u, v \in [t] : u \neq v, p \in {}^\bullet u \cap {}^\bullet v\}$

Für einen AKR-Cluster $Clust = Clust(t)$ einer beliebigen Transition t, können wir nun eine Fallunterscheidung vornehmen, welche Knoten bereits einer Komponente zugeordnet sind.

1. Für alle Knoten $x \in Clust$ gilt $\mathcal{D}(x) = \bot$, es ist also noch kein Knoten einer Komponente zugewiesen. Dann wählen wir eine neue Komponente v und weisen allen Knoten aus $Clust$ diese Komponente zu: $\mathcal{D}(x) = v$ für alle $x \in Clust$.

2. Es gibt mindestens einen Knoten $x \in Clust$, dem bereits eine Komponente $\mathcal{D}(x) \neq \bot$ zugewiesen wurde. Jeder weitere Knoten, dem eine Komponente zugewiesen wurde, wurde die gleiche Komponente zugewiesen, also $x, y \in Clust, \mathcal{D}(x) \neq \bot, \mathcal{D}(y) \neq \bot \implies \mathcal{D}(x) = \mathcal{D}(y)$. Dann bekommen alle Knoten in $Clust$ die Komponente von x zugewiesen: $\mathcal{D}(y) = \mathcal{D}(x)$ für alle $y \in Clust$.

3. Es gibt mindestens zwei Knoten $x, y \in Clust$ mit $x \neq y$, denen bereits eine Komponente $\mathcal{D}(x) \neq \bot, \mathcal{D}(y) \neq \bot$ zugewiesen wurde, jedoch unterschiedliche $\mathcal{D}(x) \neq \mathcal{D}(y)$. Da alle Knoten in $Clust$ zur gleichen Komponente gehören müssen, ergibt sich hier ein Widerspruch. Da die Verteilung von x und y bereits festgelegt ist, wir aber eine von beiden ändern müssten, gibt es für $Clust$ keine Verteilung.

Wenn wir wenigstens einen AKR-Cluster finden, bei dem die dritte Bedingung zutrifft, können wir den Adapter nicht verteilen. Dieser Fall tritt ein, wenn zwei Transformationsregeln R_i und R_j ($i \neq j$) auf unterschiedliche Komponenten verteilt werden sollen, die entsprechenden Transitionen aber im gleichen AKR-Cluster liegen $R_j \in Clust(R_i)$.

Mit dieser Beobachtung lässt sich folgendes Lemma zeigen.

Lemma 46 (Existenz einer Verteilung)
Ein Adapter lässt sich genau dann verteilen, wenn für je zwei Regeltransitionen R_i und R_j $(i \neq j)$, die auf unterschiedliche Komponenten $\mathcal{D}(R_i) \neq \mathcal{D}(R_j)$ verteilt werden sollen, gilt, dass sie nicht transitiv in Aktivierter-Konflikt-Relation stehen, also $R_i \notin Clust(R_j)$.

Lässt sich ein Netz verteilen, ordnen wir den restlichen Knoten, die außerhalb der AKR-Cluster liegen, jeweils eine eigene Komponente zu. Bei der Implementation unterscheiden wir, ob entweder zwischen zwei Transitionen eine verteilter Konflikt existiert und wählen die GGS-Implementation, oder wir wählen die einfache Implementation.

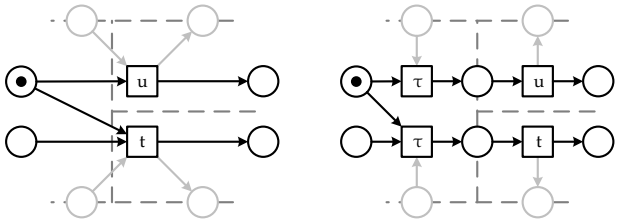
Anschließend zerschneiden wir die Implementation des Netzes entlang der eingeführten Schnittstellenplätze die Komponenten.

Theorem 47 (Maximalität der Verteilung)
Wenn eine Verteilung existiert, dann ist sie maximal.

Beweis.
Grundsätzlich sieht der Algorithmus vor, dass jeder Knoten auf eine eigene Komponente verteilt wird. Die einzige Ausnahme sind Knoten, die gemäß Definition 45 in einem AKR-Cluster liegen. Das Verteilen von Knoten in einem AKR-Cluster zerstört die Step-Readiness-Äquivalenz. Damit ist die Verteilung eines AKR-Cluster auf eine Komponente bezüglich dieser Äquivalenz maximal. Alle anderen Knoten sind maximal verteilt. ∎

6.3.2 *Spezialfall asynchrone Controllerschnittstelle*

Der vorgestellte Algorithmus zum Verteilen eines Adapters ist unabhängig davon, ob wir einen Adapter oder ein beliebiges Netz mit einer partiellen Verteilung betrachten. Für den Fall, dass die Schnittstelle zwischen Engine und Controller asynchron ist, können wir immer eine Verteilung des Adapters finden.



(a) Vor dem Verteilen (b) Nach dem Verteilen

Abbildung 39: Verteilen durch Aufspalten von Transitionen; die Schnittstelle zum Controller ist durch die grauen Plätze angedeutet

Im Fokus steht hier der Fall, dass wir jede Regeltransition auf eine eigene Komponente verteilt werden soll. So können wir zum Beispiel die Implementation einer Transformationsregel von allen anderen trennen. Allerdings wird dabei der Rest des Adapters nicht verteilt. Insbesondere liegen alle Nachrichtenplätze der Engine dann in der gleichen Komponente.

Ein Vorteil der asynchronen Schnittstelle zwischen Engine und Controller ist, dass wir den Controller bereits als eigene Komponente ansehen können – die Schnittstelle zu den Regeltransitionen ist bereits asynchron –, sodass wir uns lediglich um die Verteilung der Engine kümmern müssen.

Van Glabbeek et al. verteilen ein Netz durch das Einfügen einer τ-Transition und einem zusätzlichen Platz. Ein Konflikt wird dabei nichtdeterministisch durch die τ-Transitionen aufgelöst. In unserem Technik zur Adaptersynthese hingegen wird ein Konflikt durch den Controller aufgelöst oder gar nicht erst zugelassen.

Wir spalten eine Regeltransition in zwei Transitionen auf, sodass die erste Transition vom Controller aktiviert und die zweite die Transformationsregel umsetzt und den Controller informiert, wenn sie geschaltet hat. Der Platz zwischen diesen beiden Transitionen dient uns als Schnittstellenplatz. Ein Beispiel mit einem verteilten Konflikt sehen wir in Abbildung 39.

Links in Abbildung 39a sehen wir die Ausgangssituation. Die beiden mit u und t gelabelten Regeltransitionen sollen auf unterschiedliche Komponenten verteilt werden. Dies erreichen wir, indem wir sie jeweils auf eine andere Komponente verteilen als

ihren Vorbereich. Jede Transition besitzt zudem noch je einen hellgrauen Aktivierungs- und Informationsplatz für die Kommunikation mit dem Controller.

Aufgrund des gezeigten verteilten Konflikts können wir mit der bereits eingeführten Methode diese Netzstruktur nicht verteilen. Murata beschreibt in seinem Übersichtspapier [80] zu Petrinetzen auch eine Reihe von strukturellen Änderungen an Netzen, die Sicherheit, Lebendigkeit und Beschränktheit erhalten. Dazu gehört auch das Zusammenfassen von hintereinander liegenden Transitionen, beziehungsweise die Rückrichtung, die wir hier nutzen.

Wir dürfen eine Transition $t_{1,2}$ durch zwei Transitionen t_1 und t_2 und einen Platz p ersetzen, sodass t_1 den gleichen Vorbereich wie $t_{1,2}$ und t_2 den gleichen Nachbereich wie $t_{1,2}$ hat. Der Platz p hat t_1 als Vorbereich und t_2 als Nachbereich. Nach dem Ersetzen ist das Netz genau dann sicher, lebendig beziehungsweise beschränkt, wenn es dies vor dem Ersetzen war.

Für die Korrektheitskriterien, die wir in dieser Arbeit betrachten, ist dies ausreichend, um die Netztransformation anwenden zu können. Ein offenes Netz nach dem Ersetzen ist genau dann schwach terminierend, wenn es vor dem Ersetzen schwach terminierend war.

Wie wir diese Netztransformation ausnutzen, sehen wir in Abbildung 39b. Aus der Transition mit dem Label u in Abbildung 39a sind nun zwei Transitionen mit den Labeln ffi und u entstanden. Der Platz zwischen den beiden Transitionen dient uns als Schnittstellenplatz und wir können die beiden Transitionen auf unterschiedliche Komponenten verteilen. Die Regeltransitionen mit den Labeln u und t stehen nicht mehr im Konflikt und können auf unterschiedliche Komponenten verteilt werden.

Der Controller aktiviert die entsprechende τ-Transition und wenn die Regeltransition geschaltet hat, erhält der Controller die Information darüber. Für den Controller ist das Schalten der τ-Transition nicht beobachtbar, weshalb sich die so veränderte Engine für den Controller wie zuvor verhält.

Den ursprünglichen Konflikt zwischen den Transitionen können wir nicht auflösen, jedoch betrifft er keine Regeltransitionen mehr. Insofern können wir alle Regeltransitionen auf jeweils eine eigene Komponente verteilen und so zum Beispiel deren Implementation schützen.

6.4 Implementation

Eine Implementation entstand im Rahmen einer Studienarbeit [27] als Erweiterung zu einer bestehenden Bibliothek für offene Netze. Die Implementation nutzt jedoch ein ausschließlich strukturelles Kriterium, indem sie strukturelle Konflikte findet und diese nicht verteilt. Wir können damit weniger Adapter verteilen, als dies durch die oben gegebenen Definitionen möglich ist. Allerdings konnten wir feststellen, dass bei vielen, interessanten Fällen ein Verteilung möglich ist.

Die Einschränkung auf strukturelle Konflikte ist dadurch begründet, dass die im Rahmen der Arbeit veränderte Bibliothek Petrinetze und offene Netze strukturell unterstützt und Verhalten nur marginal betrachtet. Außerdem konnten wir feststellen, dass in unserem Fall ein struktureller Konflikt einen verteilten Konflikt impliziert. Da wir im Werkzeug zur Technik aus Kapitel 3 Transitionen des Adapters, die nicht schalten können, entfernen, ist jede Transition des Adapters aktiviertbar. Damit ist insbesondere jede im Konflikt stehende Transition aktivierbar, und die Bedingungen für einen verteilten Konflikt sind erfüllt. Was an dieser Stelle fehlt, um die Verteilung bestmöglich umzusetzen, ist die Überprüfung, ob es sogar einen aktivierten Konflikt gibt.

Für die einfache asynchrone Implementation mit einer τ-Transition und zugehörigem Platz sind somit alle Mittel vorhanden. Was fehlt, ist das Aufbauen des Erreichbarkeitsgraphen für den Adapter in Komposition mit den gegebenen offenen Netzen. Mit diesem könnten wir prüfen, ob ein verteilter Konflikt sogar ein aktivierter Konflikt ist, oder ob wir die GGS-Implementation anwenden können.

6.5 Zusammenfassung

In diesem Kapitel haben wir betrachtet, wie wir einen generierten Adapter auf Komponenten verteilen können. Ausgangspunkt dafür ist eine vorgegebene Verteilung der Transformationsregeln auf Komponenten.

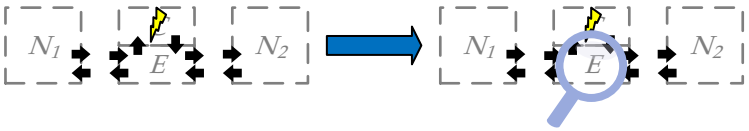
Wir bauen für die Verteilung auf Arbeiten von Glabbeek, Goltz und Schicke [40] auf. Diese beschreiben eine asynchrone Implementation basierend auf einer Verteilungsfunktion unter Bewahrung von Step-Readiness-Äquivalenz. Sie identfizieren den verteilten Konflikt als Struktur, die bezüglich der Äquivalenz nicht verteilt werden darf.

Wir gehen von einer partiellen Verteilungsfunktion aus, die nur für die Regeltransitionen vorgegeben ist. Unter Berücksichtigung von Konfliktclustern vervollständigen wir die Verteilungsfunktion auf den gesamten Adapter, sofern dies möglich ist. Die Step-Readiness-Äquivalenz bewahrt dabei die von uns betrachteten Korrektheitskriterien wie Verklemmungsfreiheit und Schwache Terminierung.

Wir bieten außerdem eine Implementation, die verteilte Konflikte in Adaptern berücksichtigt.

7 DIAGNOSE

Wenn die Adaptersynthese nicht gelingt, können wir Hinweise geben, welche Transformationsregeln hinzugefügt werden können, um die Synthese zu ermöglichen?



7.1 PROBLEMSTELLUNG

In Kapitel 3 haben wir unsere grundlegende Technik, Adaptersynthese auf Controllersynthese zurückzuführen, kennengelernt. Überlegungen in Kapitel 4 haben gezeigt, dass die Existenz eines Controllers zum einen davon abhängt, dass die gegebenen Netze kontrollierbar sind, und zum anderen von der Menge der Transformationsregeln. In diesem Kapitel betrachten wir den Fall, dass wir zwei kontrollierbare offene Netze N_1 und N_2 und eine Menge $\mathcal{R}$ von Transformationsregeln mit entsprechender Engine $E = \mathit{Engine}(N_1, N_2, \mathcal{R})$ gegeben haben, und die Controllersynthese fehlschlägt. Es existiert also kein Controller für $N_1 \oplus E \oplus N_2$.

Da die beiden gegebenen offenen Netze N_1 und N_2 kontrollierbar sind, fehlen folglich Transformationsregeln, die es uns erlauben würden, einen Adapter zu synthetisieren. Wir identifizieren Punkte im erreichbaren Verhalten der Engine, an denen das Hinzufügen einer weiteren Transformationsregel das Verhalten der Engine erweitern würde.

Die Idee, solche Punkte ausfindig zu machen basiert auf der Arbeit von Lohmann [58]. Er gibt Gründe an, warum die Synthese fehlschlagen kann. Der erste Grund sind *nicht-kommunizierte Entscheidungen* und der zweite *unvermeidbare Verklemmungen.*

Bei unserer Technik gibt es keine unkommunizierten Entscheidungen, die wir als Sündenbock bezeichnen können. Die Engine informiert nämlich den Controller über jedes Schalten einer Transition. Eines der beteiligten offenen Netz kann zwar eine interne, unkommunizierte Entscheidung treffen in Kapitel 4 haben wir festgestellt, dass die beteiligten offenen Netze selbst kontrollierbar sein müssen. Mit dieser Prämisse ist die interne Entscheidung für einen Partner, also auch den Adapter, unproblematisch.

Der andere mögliche Grund betrifft eine unvermeidbare Verklemmung. Um diese zu vermeiden, müssen wir dem Controller erlauben, mehr Verhalten zu steuern, das bisher nicht unter seiner Kontrolle war. Die gegebenen offenen Netze wollen wir aber bei unserer Technik gerade nicht verändern, und über die Engine hat eine Controller bereits komplette Kontrolle. Insofern ist mehr Kontrolle nicht möglich.

Wir konzentrieren uns daher ausschließlich darauf, mehr Transformationsregeln angeben zu können, da dies die Chancen erhöht, eine Controller und somit einen Adapter synthetisieren zu können.

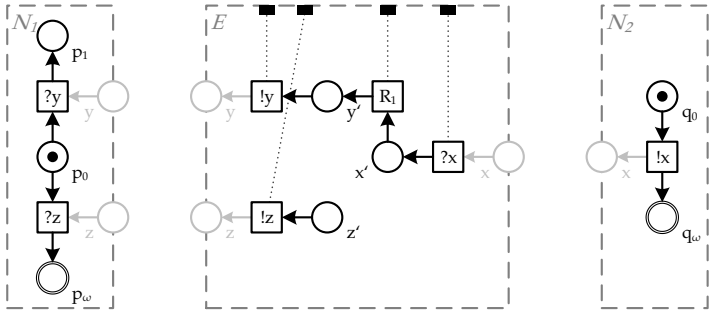
Wir betrachten ein Beispiel, um zu zeigen, dass wir aus einer Verklemmung Informationen ableiten können, die uns helfen, weitere Transformationsregeln hinzuzufügen. Diese Idee verallgemeinern wir im Laufe des Kapitels.

7.1.1 *Beispiel*

Wir betrachten das technische Beispiel in Abbildung 40, das aus zwei offenen Netzen N_1 und N_2 besteht, sowie einer Engine E, welche die Regel $\mathrm{R}_1\colon \mathrm{x} \to \mathrm{y}$ implementiert.

Das offene Netz N_1 in Abbildung 40a kann in der Anfangsmarkierung entweder eine Nachricht y empfangen und verklemmen, oder es empfängt eine Nachricht z und ist in einer Endmarkierung. Das offene Netz N_2 in Abbildung 40c sendet eine Nachricht x und ist in einer Endmarkierung. Die Engine E in Abbildung 40b implementiert die Schnittstelle zu N_1 und N_2 und die Regel $\mathrm{R}_1\colon \mathrm{x} \to \mathrm{y}$.

Aus der Startmarkierung $[p_0, q_0]$ von $N_1 \oplus E \oplus N_2$ gibt es einen Ablauf. Das offene Netz N_2 sendet die Nachricht x, die von der Engine E empfangen wird. Durch Anwenden von R_1 wird aus x ein y. Die Nachricht y wird von E gesendet und vom offenen Netz N_1 empfangen. Durch das Empfangen der Nachricht erreichen wir die Markierung $[p_1, q_\omega]$, die keine Endmarkierung ist und in der keine Transition aktiviert ist. Wir erreichen also eine Verklemmung.



(a) N_1 (b) E (c) N_2

Abbildung 40: Zwei offene Netze mit Engine, für die kein Controller existiert

In der Markierung $[p_1, q_\omega]$ können die beiden offenen Netze nicht mehr schalten. Auch die Engine kann nicht mehr schalten und somit ist es nicht sinnvoll, die Engine in dieser Situation noch anzupassen. Wir betrachten deshalb eine Situation, in der es möglich ist, dass Verhalten zu beeinflussen. In dem Zustand, in dem die Engine die Nachricht y sendet, ist es sinnvoll, stattdessen die Nachricht z zu senden. Mit der Nachricht z kann N_1 seine Endmarkierung erreichen.

Um die Nachricht z senden zu können, müssen wir sie entweder erzeugen oder aus einer anderen Nachricht transformieren. Zu dem Zeitpunkt, in dem das Senden von z sinnvoll ist, liegt entweder die Nachricht x oder die Nachricht y in der Engine vor. Drei denkbare, alternative Regeln bezüglich der Kommunikation mit N_1 sind somit: R_2: $\rightarrow$ z, R_3: x $\rightarrow$ z, R_4: y $\rightarrow$ z.

Wir führen hier einen generischen Ansatz ein. Er verallgemeinert die Idee von Lohmann [58], sodass er nicht nur für Verklemmungsfreiheit mit garantierter Kommunikation genutzt werden kann. Voraussetzung ist jedoch, dass es für das entsprechende Kriterium einen allgemeinsten Kommunikationspartner gibt, also ein Partnernetz, das alle anderen simuliert. Wir nutzen zudem aus, dass wir die Struktur des Netzes, für das wir den Controller berechnen wollen, zu einem gewissen Grad kennen. Das ermöglicht uns, die Bedeutung der Engine zur Umsetzung des Datenflusses als Teil der Diagnose einzusetzen, wohingegen der allgemeine Diagnoseansatz nicht zwischen Kontrollfluss und Datenfluss unterscheidet.

Wir betrachten im Folgenden die Diagnose exemplarisch für Schwache Terminierung. Die entsprechenden Definitionen beziehen sich jedoch auf den allgemeinen Fall, für den wir die Existenz eines allgemeinsten Kommunikationspartners voraussetzen.

AUSGANGSSITUATION Gegeben seien zwei kontrollierbare offene Netze N_1 und N_2 sowie eine Menge $\mathcal{R}$ an Transformationsregeln. Wir nehmen an, dass wir mit unserer Technik aus Kapitel 3 keinen Controller synthetisieren können und geben Möglichkeiten an, um $\mathcal{R}$ zu erweitern, sodass die Engine in Verbindung mit N_1 und N_2 mehr Verhalten zeigen kann. In der Betrachtung und den Definitionen beziehen wir uns jeweils nur auf N_1. Die Betrachtung für N_2 ist vollkommen analog.

GLIEDERUNG Wir betrachten zuerst in Abschnitt 7.2 ein Beispiel, um die Idee der Diagnose anschaulich einzuführen. Anschließend beschreiben wir in Abschnitt 7.3, welches Kommunikationsverhalten der Engine mit einem offenen Netz während der Controllersynthese erreichbar ist. Dieses vergleichen wir mit dem allgemeinsten Kommunikationspartners der offenen Netzes und leiten in Abschnitt 7.4 Diagnoseinformationen ab. Auf die Implementation der Idee gehen wir in Abschnitt 7.5 ein, bevor wir in Abschnitt 7.6 die Ergebnisse zusammenfassen.

7.2 EINFÜHRENDES BEISPIEL

Die Informationen für die Diagnose erhalten wir, in dem wir das Verhalten, das eine Engine mit einem offenen Netz N_1 zeigen kann, mit dem Verhalten, das der allgemeinste Kommunikationspartner von N_1 zeigen kann, vergleichen. Wenn der allgemeinste Kommunikationspartner mehr Verhalten zeigt als die Engine, dann ist dies eine Situation, an der wir das Verhalten der Engine sinnvoll ändern könnten.

Abbildung 41 veranschaulicht diese Idee. Linker Hand sehen wir den allgemeinsten Kommunikationspartner für das offene Netz N_1 aus Abbildung 40a. Vom Anfangszustand s_0 kann er ein z senden und ist im Endzustand s_1. Rechter Hand sehen wir das Kommunikationsverhalten der Engine E mit N_1. Aus dem Anfangszustand b_0 geht sie durch Senden der Nachricht y in den Zustand b_1 über, von dem aus kein weiteres Kommunikationsverhalten möglich ist.

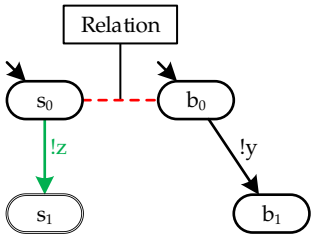


Abbildung 41: Relation zwischen allgemeinstem Kommunikationspartner von N_1 und dem Kommunikationsverhalten von E mit N_1

Um nun alternatives Verhalten zu bestimmen, das im allgemeinsten Kommunikationspartner, aber nicht im beobachtetem Kommunikationsverhalten zwischen E und N_1 vorhanden ist, betrachten wir die Teile von E und N_1, die in Simulationsbeziehung stehen. In Abbildung 41 ist dies durch die rote gestrichelte Linie angedeutet. Die beiden Anfangszustände s_0 und b_0 sind in Relation.

Für in Relation stehende Zustände können wir vergleichen, ob im Zustand des allgemeinsten Kommunikationspartner Verhalten möglich ist, das im Zustand des Kommunikationsverhalten der Engine nicht möglich ist. Im Beispiel betrifft dies den mit !z beschrifteten Übergang in s_0. Bei dem in Relation zu s_0 stehenden Zustand b_0 gibt es solchen einen Übergang nicht. Das Senden einer Nachricht z wäre somit mögliches alternatives Verhalten, das in b_0 mit der gegeben Engine E nicht möglich ist.

Wir hatten bereits im Beispiel in Abbildung 40 gesehen, dass die Engine anstelle der Nachricht y die Nachricht z senden müsste, damit das offene Netz N_1 seine Endmarkierung erreichen kann. Eine zentrale Frage ist, wie wir erreichen, dass die Engine die Nachricht z senden kann.

Um eine Nachricht senden zu können, müssen wir diese in der Engine generieren können – entweder aus dem Nichts oder aus anderen Nachrichten. Aufgrund der semantischen Bedeutung einer Nachricht, ist es in der Regel nicht sinnvoll, sie ohne Kontext aus dem Nichts zu erzeugen – obwohl uns das die Existenz eines Adapters garantieren würde, wie in Kapitel 4 gesehen. Die Alternative ist somit, die Nachricht aus anderen, bereits vorhandenen Nachrichten zu erzeugen.

Dabei greifen wir nur auf Nachrichten zurück, die im Zustand b_0 unseres Beispiels vorhanden sind, also bevor die Engine die Nachricht y sendet. In der Menge von

Markierung, die b_0 umfasst, gibt es Markierungen, in denen entweder die Nachricht x oder die Nachricht y in der Engine vorhanden ist.

Wenn wir annehmen, dass die Nachricht z semantisch korrekt aus der Nachricht x erzeugt werden kann, können wir eine weitere Transformationsregel R_2: x → z zur Menge der Transformationsregeln hinzufügen. Wenn wir mit dem nun größeren Satz an Regeln die Synthese wiederholen, kann die Engine neues Kommunikationsverhalten mit N_1 zeigen und wir finden tatsächlich einen Adapter mit Hilfe dieser zweiten Regel.

7.3 KOMMUNIKATIONSVERHALTEN DER ENGINE MIT EINEM OFFEN NETZ

Die zentrale Idee unseres Ansatzes besteht darin, das mögliche Kommunikationsverhalten der Engine mit einem offenen Netz N_1 zu beschreiben. Bei der Controllersynthese wird der Raum der erreichbaren Markierungen in der Komposition von $N_1 \oplus E \oplus N_2 \oplus U$ betrachtet, wobei U die universelle Umgebung ist. Danach abstrahieren wir das Transitionssystem mit Definition 17 auf die Transitionen von U, um das Verhalten aller möglichen Controller zu überapproximieren. Analog können wir mit den Transitionen verfahren, die in der Engine für die Kommunikation mit N_1 zuständig sind.

Die Komposition $N_1 \oplus E \oplus N_2 \oplus U$ enthält auch das gesamte mögliche Kommunikationsverhalten der Engine E mit dem offenen Netz N_1. Dazu müssen wir eben jene Transitionen $T_{E \to N_1}$ von E betrachten, die mit N_1 kommunizieren.

Definition 48 ($T_{E \to N_1}$)
Die Menge $T_{E \to N_1}$ umfasst alle Transitionen einer Engine E, welche für die Kommunikation mit dem offenen Netz N_1 zuständig sind – also ein entsprechendes Kommunikationslabel besitzen:

$$T_{E \to N_1} = \{t \in T_E \mid \lambda_E(port_{N_1}, t) \neq N_1.\tau\}$$

Anstatt nun den Erreichbarkeitsgraphen $\Re(N_1 \oplus E \oplus N_2 \oplus U)$ bezüglich der Transitionen von U zu abstrahieren und zu beschränken, abstrahieren wir den Erreichbarkeitsgraphen bezüglich der Transitionen $T_{E \to N_1}$ gemäß Definition 17.

Definition 49 (Kommunikationsverhalten der Engine *E*)
Wir erhalten das *Kommunikationsverhalten der Engine E*, indem wir den Erreichbarkeitsgraphen $\Re(N_1 \oplus E \oplus N_2 \oplus U)$ auf das Verhalten $\mathcal{B}$ der Transitionen $T_{E \to N_1}$ abstrahieren (Definition 17):

$$\mathcal{B} = \mathcal{B}(\Re(N_1 \oplus E \oplus N_2 \oplus U), T_{E \to N_1})$$

7.4 Ableiten der Diagnoseinformationen

Um die Diagnoseinformationen abzuleiten, setzen wir den allgemeinsten Kommunikationspartner mit dem Kommunikationsverhalten der Engine in Simulationsbeziehung. So bestimmen wir, wo sich beide gleich verhalten, und insbesondere, wo sie sich unterschiedlich verhalten.

Definition 50 (Simulationsbeziehung)
Seien $\mathcal{A}$ und $\mathcal{B}$ zwei Transitionssysteme. Wir definieren eine Simulationsbeziehung $\rho \subseteq Q_{\mathcal{A}} \times Q_{\mathcal{B}}$ rekursiv wie folgt:

- $\langle q_{\mathcal{A}}, q_{\mathcal{B}} \rangle \in \rho$, falls $q_{\mathcal{A}}$ der Startzustand von $\mathcal{A}$ und $q_{\mathcal{B}}$ der Startzustand von $\mathcal{B}$ ist, und
- falls $\langle q_{\mathcal{A}}, q_{\mathcal{B}} \rangle \in \rho$ und es existieren Übergänge $\left\langle q_{\mathcal{A}}, t_{\mathcal{A}}, q'_{\mathcal{A}} \right\rangle$ und $\left\langle q_{\mathcal{B}}, t_{\mathcal{B}}, q'_{\mathcal{B}} \right\rangle$ mit $\lambda(t_{\mathcal{A}}) = \lambda(t_{\mathcal{B}})$, dann $\left\langle q'_{\mathcal{A}}, q'_{\mathcal{B}} \right\rangle \in \rho$.

Klassischerweise benutzen wir eine Simulationsbeziehung zwischen zwei Systemen, um auszudrücken, dass ein System ein anderes System *vollständig* simuliert – das zweite System also jeden Schritt des ersten Systems nachahmen kann. Wir interessieren uns nur dafür, wann sich das zweite System wie das erste verhält. Denn die Zustände in denen das erste System Verhalten zeigt, das vom zweiten System nicht simuliert werden kann, bieten uns Diagnoseinformationen. In diesem Sinne unterscheidet sich unsere Definition von der klassischen, in der zwei Systeme nur dann in einer Simulationsbeziehung stehen, wenn ein System das andere vollständig simuliert. In unserer

Definition betrachten wir lediglich die maximalen Teile der Systeme, die in Simulation stehen. Im einfachsten Fall stehen nur die beiden Anfangszustände in Beziehung.

Die von uns betrachteten Transitionssystem sind deterministisch, daher ist die Simulationsbeziehung in Definition 50 eindeutig.

Lemma 51
Die Simulationsbeziehung ρ ist eindeutig definiert, wenn die beiden in Beziehung zu setzenden Systeme deterministisch sind.

Beweis.
Die Aussage folgt sofort aus der Voraussetzung des Determinismus, da es in jedem Knoten für jedes Label maximal eine ausgehende Kante gibt. Damit steht der entsprechende Nachfolgeknoten entweder nicht oder mit genau einem Knoten im anderen System in Beziehung, da dieses ebenfalls deterministisch ist. Dieses Argument lässt sich induktiv auf das gesamte System fortsetzen. ∎

Ein weiterer Unterschied unserer Definition zur klassischen Simulationsbeziehung ist die fehlende Asymmetrie in der Definition. Dies liegt zum einen daran, dass wir nicht das vollständige Verhalten eines Systems in Beziehung stellen. Zum anderen aber auch daran, dass wir deterministische Systeme betrachten. Es gibt weder τ-Übergänge noch verschiedene von einem Knoten ausgehenden Kanten mit dem gleichen Label. Die Asymmetrie entsteht allein durch den Ursprung und somit der Bedeutung der Transitionssysteme, die wir in Beziehung setzen.

Wir möchten wissen, welches zusätzliche Verhalten im allgemeinsten Kommunikationspartner möglich ist, das nicht im Kommunikationsverhalten der Engine auftritt. Insofern ist es wichtig zu unterscheiden, zu welchem System eine Kante gehört. Im Folgenden sei daher $\mathcal{A}$ der allgemeinste Kommunikationspartner des offenen Netzes N_1 und $\mathcal{B}$ das mögliche Kommunikationsverhalten der Engine E mit N_1.

Zusätzliches Verhalten entspricht dann Kommunikation, die in einem Zustand von $\mathcal{A}$ möglich ist, aber im in Relation stehenden Zustand von $\mathcal{B}$ nicht möglich ist. Hier reicht es sogar aus, nur Sendekanäle zu betrachten; den Empfang einer Nachricht kann die Engine nicht erzwingen, weil die Nachricht vom offenen Netz stammt. Das Senden kann die Engine jedoch beeinflussen.

Definition 52 (Zusätzliches Verhalten)
Seien $\mathcal{A}$ der allgemeinste Kommunikationspartner des offenen Netzes N_1, $\mathcal{B}$ das Kommunikationsverhalten der Engine E mit N_1 und ρ die Simulationsbeziehung zwischen $\mathcal{A}$ und $\mathcal{B}$. Dann definieren wir *zusätzliches Verhalten* plus als ein Prädikat über den Zuständen in $\mathcal{B}$ und den Sendekanälen. Das Prädikat $\text{plus}(q_{\mathcal{B}}, l)$ gilt genau dann, wenn es einen zu $q_{\mathcal{B}}$ in Simulationsbeziehung stehenden Zustand in $\mathcal{A}$ gibt, in dem es keine ausgehende Kante mit dem Label l gibt.
$\text{plus}(q_{\mathcal{B}}, l)$ gdw. $\exists \langle q_{\mathcal{A}}, q_{\mathcal{B}} \rangle \in \rho$ mit $q_{\mathcal{A}} \xrightarrow{t_{\mathcal{A}}} q'_{\mathcal{A}}$ und $\lambda(t_{\mathcal{A}}) = l$, sodass $q_{\mathcal{B}} \not\xrightarrow{t_{\mathcal{B}}}$ mit $\lambda(t_{\mathcal{B}}) = l$.
Die Menge allen möglichen zusätzlichen Verhaltens ist dann

$$\text{plus}(\mathcal{B}) = \{\langle q_{\mathcal{B}}, l \rangle \mid \text{plus}(q_{\mathcal{B}}, l)\}$$

Mit Hilfe dieser Definition bestimmen wir, welche Nachricht l das offene Netz N_1 in einem bestimmten Zustand zu empfangen bereit wäre, jedoch nicht erhält. Das Senden von l in diesem Zustand erlaubt E somit mehr Kommunikationsverhalten.

Wenn wir nun jedoch wieder die semantische Bedeutung der Engine betrachten, dann ist klar, dass wir solch ein l in der Regel nicht einfach erzeugen dürfen. Semantisch sinnvoll ist es, l aus anderen Nachrichten zu transformieren.

Als Kandidaten, aus denen wir l erzeugen können, dienen uns die Nachrichten, die in dem entsprechenden Zustand in der Engine E zur Verfügung stehen. Der Zustand $q_{\mathcal{B}}$ umfasst alle Markierungen, in denen sich das Gesamtsystem befinden kann. Jede einzelne Markierung bietet somit eine Grundlage, Nachrichten, die sich gerade in der Engine befinden, zu nutzen, um l zu erzeugen.

Definition 53 (Verfügbare Nachrichten)
Sei $q_{\mathcal{B}}$ ein Zustand von $\mathcal{B}$ und somit eine Menge von Markierungen. Dann sind die *verfügbaren Nachrichten* die in $q_{\mathcal{B}}$ enthaltenen Markierungen eingeschränkt auf die Engine:

$$\text{pending}(q_{\mathcal{B}}) = \{\mu_{|E} \mid \mu \in q_{\mathcal{B}}\}$$

Wenn es nun verfügbare Nachrichten $\mu \in \text{pending}(q_{\mathcal{B}})$ gibt, aus denen wir die Nachricht l erzeugen können, erstellen wir eine neue Transformationsregel, die das Kommunikationsverhalten zwischen Engine E und offenem Netz N_1 vergrößert.

Alternativ kann die benötigte Nachricht l mit Hilfe einer Transformationsregel erzeugt werden, die zum Erreichen des Zustandes $q_{\mathcal{B}}$ angewendet wurde, d. h., die Nachricht l wird früher erzeugt und später in dem entsprechenden Zustand gesendet. Dabei ziehen wir alle Regeln in Betracht, die zum Erreichen eines entsprechenden Zustandes führen.

Definition 54 (Angewendete Transformationsregeln)
Die Menge der angewendeten Transformationsregeln in einem Zustand $q_{\mathcal{B}}$ ergibt sich aus allen Sequenzen, die vom Startzustand q_0 von $\mathcal{B}$ zu $q_{\mathcal{B}}$ führen und den dort entsprechend angewendeten Regeltransitionen.

$$\text{rules}(q_{\mathcal{B}}) = \{\text{R} \mid \exists \vec{t} = \ldots, t_{\text{R}}, \ldots : q_0 \stackrel{\vec{t}}{\Rightarrow} q_{\mathcal{B}}\}.$$

Somit haben wir alle Diagnoseinformationen zusammen: Zum einen bestimmen wir Zustände, in denen die Engine weniger Verhalten im Vergleich zum allgemeinsten Kommunikationspartner zeigt. Zum anderen definieren wir die beiden Möglichkeiten, um eine fehlende Nachricht zu erzeugen, nämlich im entsprechenden Zustand in der Engine vorhandene Nachrichten oder Regeln, die zu dem Zustand führen.

Bis hierhin haben wir nur betrachtet, wie wir der Engine mehr Verhalten ermöglichen. Allerdings gibt es einen Sonderfall, in dem weiteres Senden von Nachrichten nicht sinnvoll ist. Wenn die gegebenen offenen Netze bereits jeweils in einer Endmarkierung sind, die Engine jedoch nicht, dann bedeutet dies, dass es überflüssige Nachrichten gibt, die wir entfernen müssen. Bei elektronischen Nachrichten ist die einfachste Option, die Nachrichten mit einer entsprechenden Transformationsregel zu löschen. Unter Umständen spricht die semantische Bedeutung einer Nachricht dagegen, sie zu löschen, weil sie eine wichtige Information trägt; der Sender erwartet, dass die Nachricht empfangen wird. Die Alternative ist, diese Nachricht gar nicht erst zu erzeugen. Hier können wir wieder die Regeln betrachten, die zu dem entsprechenden Zustand der Engine führen, und eine Regel, welche die Nachricht erzeugt so abändern, dass die Nachricht nicht mehr erzeugt wird.

Definition 55 (Überflüssige Nachricht)
Ein Nachricht l ist *überflüssig*, wenn es Markierung gibt, in der die gegebenen offenen Netze jeweils in einer Endmarkierung sind, l jedoch noch in der Engine verfügbar ist.

Die so erhaltenen Informationen sind vollständig.

Lemma 56 (Vollständigkeit der Diagnoseinformationen)
Im Fall der Nichtexistenz eines Controllers, geben das zusätzliche Verhalten aus Definition 52 und die überflüssigen Nachrichten aus Definition 55 alle relevanten Informationen an, um aus diesen eine Transformationsregel zu bestimmen, die der Engine mehr Verhalten in Interaktion mit den gegebenen offenen Netzen erlaubt.

Beweis.
Ein Adapter muss korrekt mit den gegebenen offenen Netzen kommunizieren. Bei der Controllersynthese des Adapters wird inkorrektes Verhalten automatisch ausgeschlossen, d. h., insbesondere Kommunikationsverhalten der Engine, das über das Verhalten der allgemeinsten Kommunikationspartner hinaus geht. Bezüglich der offenen Netz brauchen wir somit nicht betrachten, wie wir Verhalten verhindern, sondern nur, wie wir zusätzliches Verhalten hinzufügen können. Der allgemeinste Kommunikationspartner gibt vor, welches das maximale Verhalten ist, das ein Kommunikationspartner wie der Adapter zeigen darf. In Zuständen, in denen die Engine weniger Verhalten zeigt, geben wir entsprechende Informationen gemäß Definition 52 an.

Um eine Nachricht zu erzeugen gibt es drei Alternativen: Die Nachricht lässt sich aus dem Nicht erzeugen, die Nachricht lässt sich aus den in einem Zustand verfügbaren Nachrichten der Engine erzeugen, oder die Nachricht lässt sich mit Hilfe einer bereits angewendeten Regel erzeugen. Die erste Alternative benötigt keine weiteren Informationen, für die nächsten beiden Alternativen haben wir die Definitionen 53 und 54 angegeben. Damit betrachten wir jede Nachricht, die bis zum Erreichen eines Zustandes jemals in der Engine verfügbar war, und entweder in einer Regel benötigt wurde oder noch verfügbar ist. Nur aus diesen können wir die benötigte Nachricht erzeugen.

Den Sonderfall, dass die offenen Netze in einer Endmarkierung sind, das Gesamtsystem aber nicht, betrachten wir in Definition 55. ∎

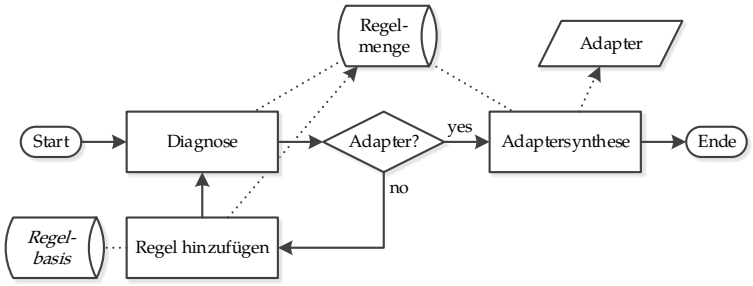


Abbildung 42: Iteratives Hinzufügen von Transformationsregeln auf der Basis von Diagnoseinformationen

7.5 IMPLEMENTATION

Die oben definierten Diagnoseinformationen sind im Adaptersynthesewerkzeug MARLENE [36] implementiert. In einem speziellen Diagnosemodus analysiert MARLENE das erreichbare Kommunikationsverhalten der Engine und gibt entsprechende Diagnoseinformationen aus.

Anhand dieser Informationen können wir manuell weitere Transformationsregeln hinzufügen. Falls die Menge der Diagnoseinformationen zu groß wird, können wir auf Techniken des Semantic Web zurück greifen, wie wir es bereits in der Einleitung für das Bestimmen von Transformationsregeln beschrieben haben.

Zur Validierung der Idee gibt es in MARLENE einen Modus, in dem aus einer Regelbasis Transformationsregeln gewählt werden können, um dann abermals Diagnose durchzuführen. Für Beispiele, für die wir die Existenz eines Adapters mit einem entsprechenden Regelsatz kennen, legen wir die Transformationsregeln zuerst in der Regelbasis ab. Die Menge der Transformationsregeln, die als Eingabe dient, wird iterativ ausschließlich auf Basis der Diagnoseinformationen erweitert. In Abbildung 42 sehen wir ein entsprechendes Flussdiagramm, das die Validierung illustriert.

Bei allen Beispielen, die in MARLENE enthalten sind, wird die vollständige Menge an Transformationsregeln nach wenigen Iterationen gefunden, sodass wir für das entsprechende Beispiel den Adapter synthetisieren können. Bei Beispielen mit Nebenläufigkeit oder Alternativen werden in einer Iteration oft mehrere Transformationsregeln hinzugefügt, da es entsprechend mehr Diagnoseinformationen pro Iteration gibt.

7.6 Zusammenfassung

In diesem Kapitel haben wir betrachtet, welche Diagnoseinformationen wir ableiten können, falls die Controllersynthese für den Adapter fehlschlägt. Wir nutzen dabei aus, dass wir die spezielle Struktur des offenen Netzes kennen, für das wir die Controllersynthese durchführen.

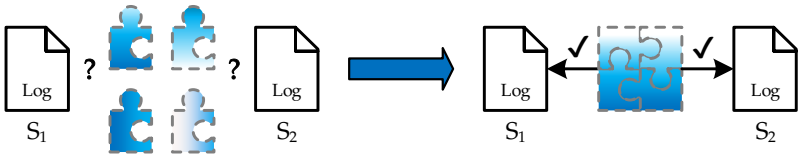
Aufbauend auf der Annahme, dass wir bei der Adaptersynthese die gegebenen offenen Netze unverändert lassen, konzentriert sich die Diagnose auf das Verhalten der Engine, die wir über die Menge der Transformationsregeln direkt beeinflussen können. Da ein Adapter ein korrekter Kommunikationspartner für jedes der gegebenen offenen Netze sein muss, vergleichen wir das Kommunikationsverhalten der Engine mit dem jeweils allgemeinsten Kommunikationspartnern der gegebenen offenen Netze.

Wir bestimmen genau die Punkte, an denen mehr Kommunikationsverhalten der Engine das Korrektheitskriterium nicht verletzt und geben die Nachrichten an, mit Hilfe derer die Kommunikation realisiert werden kann.

Die Implementation des Ansatzes erfolgte im Werkzeug Marlene für die Korrektheitskriterien Schwache Terminierung und Verklemmungsfreiheit. Aufgrund der obigen Definitionen, die vom Korrektheitskriterium abstrahieren, lässt sich der Ansatz übertragen auf Fälle, in denen wir das größte mögliche korrekte Kommunikationsverhalten mit einem offenen Netz darstellen können.

8 DISCOVERY

Wie können wir aus einer Menge an Adapationspatterns und aufgezeichnetem Verhalten ein formales Modell eines Adapters finden, welches das aufgezeichnete Verhalten am besten erklärt?



8.1 PROBLEMSTELLUNG

Nachdem wir in Kapitel 7 gesehen haben, inwieweit wir fehlende Informationen ergänzen können, betrachten wir in diesem Kapitel den Fall, dass wir keine formalen Modelle gegeben haben. Wir betrachten offene Systeme S_1 und S_2, die über eine Adapterimplementation miteinander kommunizieren. Dabei ist es nicht unwahrscheinlich, dass S_1 und S_2 ihre Ausführung in einem *Log* dokumentieren.

Für die Adapterimplementation nehmen wir weder ein formales Modell noch eine Aufzeichnung an. In der Einleitung haben wir gesehen, dass der Austausch von Nachrichten zwischen offenen Systemen konzeptionell mit Patterns beschrieben werden kann. Oft geht dies mit einer direkten Implementierung ohne vorheriger Formalisierung des Adapters einher [44].

Trotzdem möchten wir Aussagen über den Adapter treffen. Dafür müssen wir ein formales Modell erzeugen, welche das aufgezeichnete Verhalten der offenen Systeme möglichst gut erklärt. Wenn ein Modell das aufgezeichnete Verhalten gut erklärt, ist

es wahrscheinlich, dass das Modell und die laufende Implementation des Adapters gleiche, oder zumindest ähnliche Eigenschaften besitzen.

Diese Art der Fragestellung stammt aus dem Bereich des *Process Mining* [2], genauer der *Process Discovery*: Gegeben sei ein Log, das für ein implementiertes und laufendes System dessen Verhalten aufzeichnet. In der Process Discovery möchten wir ein Modell des Systems finden, welches im Log aufgezeichnetes Verhalten möglichst gut erklärt. Die Güte des Modells hängt von unterschiedlichen Kriterien ab, zum Beispiel: wie viel Verhalten vom Modell finden wir im Log und umgekehrt; die Größe des Modells.

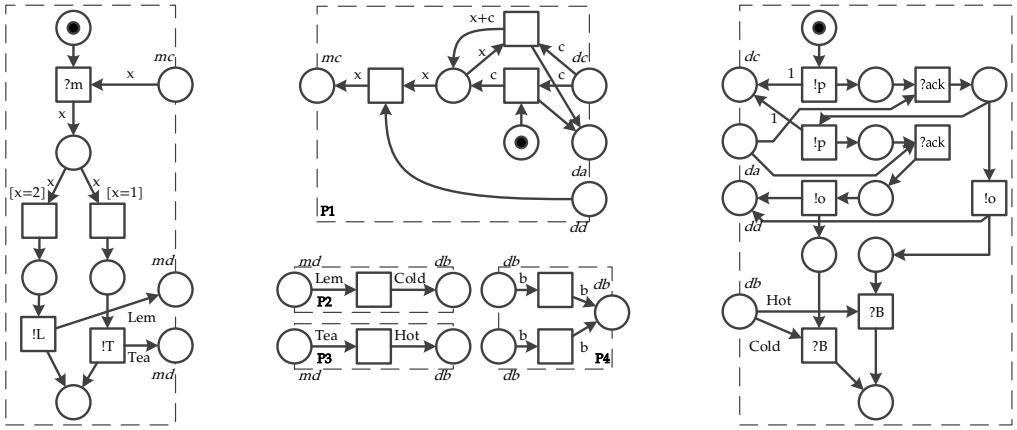
Wir betrachten einen Ansatz [38], der als Ausgangspunkt eine Menge von Patterns betrachtet. Für die Patterns nehmen wir an, dass sie in der Implementation des Adapters benutzt wurden. Konzeptionell beschränken wir uns auf die *Enterprise Integration Patterns* [44]. Formal betrachten wir jedoch Patterns, die als *Coloured Petri Nets* [47] modelliert wurden. Aufgrund der Ausdrucksmächtigkeit von Coloured Petri Nets können wir die konzeptionellen Patterns mit Coloured Petri Nets beschreiben [31].

Die alternative Idee, aus den Logs zuerst formale Modelle als offene Netze abzuleiten und für diese den in Kapitel 3 vorgestellten Ansatz zur Adaptersynthese zu nutzen, verfolgen wir hier nicht. Davon abgesehen, dass für diese Idee bereits viele Techniken existieren, besteht die Gefahr, dass wir zwei formale Modelle ableiten, die verhaltensinkompatibel sind, sodass wir nicht mal einen Adapter generieren können. Dies steht im Widerspruch dazu, dass die offenen Systeme offensichtlich über einen Adapter kommunizieren. Daher birgt diese Idee die Gefahr, kein sinnvolles Modell ableiten zu können. Wir betrachten daher einen Weg, der immer zu einem formalen Modell führt.

Als Beispiel benutzen wir eine Variante des Getränkeautomaten, der nicht aufgrund eines Tastendrucks das Getränk ausgibt, sondern aufgrund der Höhe des gezahlten Betrages. Das Beispiel sehen wir in Form von gefärbten Petrinetzen in Abbildung 43.

Der Getränkeautomat in Abbildung 43a bietet Tee oder Limonade an. Zuerst empfängt er über den Kanal ?m einen Betrag x. Wenn der Betrag 2 ist, dann gibt er im linken Zweig Limonade !L zurück. Ist der Betrag 1, dann gibt er im rechten Zweig Tee !T zurück.

Der Kunde in 43c wirft zuerst eine Münze ♭ ein, was durch ?ack bestätigt wird, und entscheidet nichtdeterministisch, ob er noch eine zweite Münze einwirft. Mit der Nachricht ♭ schließt er den Vorgang ab und erwartet das entsprechende Getränk über den Kanal B.



(a) Getränkeautomat (b) Adapterpatterns (c) Kunde

Abbildung 43: Services and Patterns

Die Patterns P1 bis P4 in Abbildung 43b ergeben den Adapter zwischen Getränkeautomat und Kunde. Im Beispiel verschmelzen wir genau die benachbarten Plätze, die auf den gestrichelten Linien jeweils nebeneinander liegen, um das Gesamtsystem zu erhalten. Das Pattern P1 fasst mehrere Münzen des Kunden zu einem Betrag zusammen, und gibt den Gesamtbetrag an den Getränkeautomaten weiter. Die beiden Patterns P2 und P3 geben die Limonade als Kaltgetränk und den Tee als Heißgetränk weiter. Das Pattern P4 leitet die beiden möglichen Getränke an den Kunden weiter.

In Komposition von Getränkeautomat und Kunde mit den vier Patterns erhält der Kunde für eine Münze den Tee als Heißgetränk und für zwei Münzen die Limonade als Kaltgetränk.

Auch wenn wir formale Modelle für den Getränkeautomaten und den Kunden zur Veranschaulichung eingeführt haben, gehen wir für den Rest des Kapitels davon aus, dass diese nicht zur Verfügung stehen, sondern ausschließlich deren aufgezeichnetes Verhalten. Die Patterns hingegen nutzen wir, um den Raum möglicher Adapter zu charakterisieren.

Die möglichen Modelle des Adapters erhalten wir, indem wir erschöpfend alle Kombinationen der gegebenen Patterns betrachten. Wir definieren dann Maße für die

Fitness und die Einfachheit eines Adapters, um so das Modell zu bestimmen, welches das aufgezeichnete Verhalten am besten erklärt.

AUSGANGSSITUATION Gegeben seien zwei Logs L_1 und L_2, welche jeweils eine Menge von Abläufen der offenen Systeme S_1 und S_2 enthalten. Wir nehmen an, dass S_1 und S_2 über einen Adapter kommunizieren. Für diesen Adapter bestimmen auf Basis einer gegebenen Menge an Patterns ein formales Modell, welches das aufgezeichnete Verhalten in L_1 und L_2 möglichst gut erklärt.

GLIEDERUNG Wir führen zuerst grundlegende Konzepte des Process Mining in Abschnitt 8.2 ein. Anschließend betrachten wir Coloured Petri Nets in Abschnitt 8.3. In Abschnitt 8.4 beschreiben wir, wie wir aus Patterns einen Adapterkandidaten erzeugen. Dessen Güte bestimmen wir in Abschnitt 8.5. Die Implementation des Ansatzes erläutern wir in Abschnitt 8.6. Wir schließen das Kapitel in Abschnitt 8.7 ab.

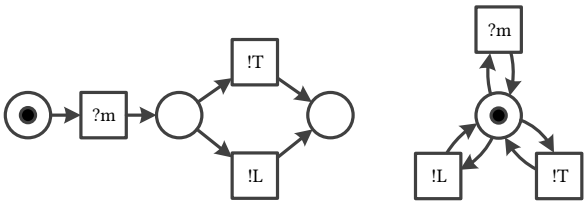
8.2 PROCESS MINING

Grundlage für das Process Mining bilden Logs. Ein Log beinhaltet eine Folge von Abläufen. Jeder Ablauf selbst besteht aus einer Sequenz von Ereignissen. Bei der Process Discovery haben wir ein Log als Eingabe und versuchen ein Modell eines Systems zu finden, welches das aufgezeichnete Verhalten gut erklärt.

Ein *Ereignis ev* besitzt für gewöhnlich verschiedene Attribute: Der *Typ* gibt an, welche Art von Ereignis aufgetreten ist, ein *Zeitstempel*, wann das Ereignis aufgetreten ist. Weitere mögliche Attribute sind zum Beispiel Datenwerte, Kommunikationsinstanzen und so weiter. Die tatsächlich für eine Ereignis verfügbaren Attribute hängen vom Kontext ab, in dem das Log erstellt wurde.

Ein *Ablauf tr* beschreibt eine Sequenz von Ereignissen einer Instanz des Systems. Innerhalb eines Ablauf sind Ereignisse chronologisch gemäß der Zeitstempel geordnet.

Als Beispiel betrachten wir in diesem Kapitel abermals einen Getränkeautomaten. Der Automat bietet Tee für einen Betrag von 1 an oder Limonade für einen Betrag von 2. Über den Betrag kann ein Kunde steuern, welches Getränk er haben möchte. Wir nehmen an, dass die folgenden beiden Abläufe im Getränkeautomaten möglich sind und im Log L aufgezeichnet wurden:



(a) α-Algorithmus (b) Blumenmodell

Abbildung 44: Mögliche Ergebnisse der Process Discovery

- $tr_1 = \langle ?m, time = 3, amount = 1\rangle, \langle !T, time = 6, bev = Tea\rangle$ und
- $tr_2 = \langle ?m, time = 7, amount = 2\rangle, \langle !L, time = 9, bev = Lemonade\rangle$.

Im ersten Fall (tr_1) empfängt der Automat eine Nachricht m zum Zeitpunkt 3 mit dem Betrag 1. Anschließend sendet er die Nachricht T zum Zeitpunkt 6 mit der Information, dass das Getränk bev Tee ist. Im zweiten Fall (tr_2) empfängt der Automat eine Nachricht m zum Zeitpunkt 7 mit dem Betrag 2. Anschließend sendet er die Nachricht L zum Zeitpunkt 9 mit der Information, dass es sich beim Getränk bev um Limonade handelt.

Je nach Präferenz eines Anwenders bietet der Bereich der Process Discovery verschiedenste Methoden an, um einen Prozess zu beschreiben, der das oben angegebene Log L erklären kann. Zwei Mögliche Prozesse sehen wir in Abbildung 44.

Der *α-Algorithmus* [2] gehört zu den bekanntesten Algorithmen in der Process Discovery. Dieser analysiert kausale Abhängigkeiten zwischen Ereignissen in einem Log und liefert ein Petrinetz als Ergebnis. Für unser Log L sehen wir das Ergebnis in Abbildung 44a. In dem gezeigten Netz kann zuerst die Transition mit dem Label ?m schalten und anschließend entweder die Transition mit Label ‼T oder die Transition mit Label ‼L. Das Netz kann also genau die beiden oben angegebenen Abläufe mit ihren Aktivitäten erzeugen. Dieses Modell kommt damit dem aus Abbildung 43a schon sehr nah.

Eine weitere mögliche Lösung ist das *Blumenmodell* in Abbildung 44b. Auch in diesem Petrinetz sind die angegebenen Abläufe möglich – aber auch beliebige andere.

An den beiden Modellen erkennen wir, dass es grundsätzlich nicht *die eine* Lösung in der Process Discovery gibt, sondern dass unterschiedliche Modelle als Lösung in Frage kommen. Eine Annahme, die wir implizit in beiden Fällen vorausgesetzt haben,

ist, dass alle im Log vorgegebenen Abläufe im Modell möglich sind. Ebenfalls können wir fordern, dass ein Modell strukturell möglichst einfach sein soll, wie zum Beispiel das Blumenmodell.

Die Güte eines Modells in Bezug auf ein Log bezeichnen wir als *Konformanz*. In der Process Discovery haben sich vier allgemeine Dimensionen herauskristallisiert [18, 94]:

FITNESS gibt an, inwieweit ein Prozessmodell die Abläufe eines Logs wiedergibt. Die Fitness ist optimal, wenn alle Abläufe im Log auch im Prozessmodell möglich sind.

PRÄZISION gibt an, wie eng ein Prozessmodell an den Abläufen, die durch ein Log vorgegeben sind, dran bleibt. Die Präzision ist optimal, wenn das Prozessmodell nur die Abläufe in den Logs hat.

EINFACHHEIT ist ein strukturelles Kriterium. Von einem strukturell einfachen Prozessmodell versprechen wir uns, es leichter zu verstehen. Typischerweise fließt die Anzahl der verwendeten Modellierungselemente und der Verzweigungen in diese Dimension ein.

GENERALISIERUNG gibt an, inwieweit ein Prozessmodell das aufgezeichnet Verhalten verallgemeinert. Da ein Log letztlich nur einem endlichen Ausschnitt des möglichen Systemverhalten entspricht, soll die Generalisierung ausdrücken, in welchem Maß Abläufe über das aufgezeichnete Verhalten hinaus möglich sind.

Die beiden Modelle (Abbildung 44a und Abbildung 44b) haben bezüglich des gegebenen Logs perfekte Fitness, da in ihnen jeweils jeder Ablauf des Logs möglich ist.

Die Präzision ist jedoch im Ergebnis des α-Algorithmus wesentlich besser als beim Blumenmodell, das dafür genereller ist. Im Blumenmodell sind beliebige weitere Abläufe möglich, wohin gegen das Ergebnis des α-Algorithmus genau die Abläufe des Logs zulässt.

Während wir bei Fitness und Präzision intuitiv die beiden Modelle noch einordnen können, ist dies bei der Einfachheit nicht mehr möglich. Ob ein Modell als einfach betrachtet wird, hängt stark von der Art der Modelle und deren Kontext ab [29, 32], und ist somit zumindest teilweise subjektiv. Oft schlägt sich Einfachheit als strukturelle

Eigenschaft nieder, zum Beispiel in der Anzahl der Plätze in einem Petrinetz oder der Anzahl ausgehender Kanten. Im ersten Fall würden wir wohl das Blumenmodell als einfacher betrachten.

Bei der Generalisierung geht es darum, wie viel zusätzliches, nicht im Log enthaltenes Verhalten wir in einem Prozessmodell beobachten können. Diese Dimension wird oft darüber quantifiziert, welches Verhalten möglich ist, nachdem wir einen Teil eines Ablaufes gesehen haben, das nicht durch das Log abgedeckt ist. Im Beispiel ist das Blumenmodell wesentlich genereller als das Prozessmodell, das wir mit dem α-Algorithmus erhalten haben.

Ein Anwender gewichtet diese vier Dimensionen entsprechend seiner Präferenzen, sofern sie Einfluss auf den Discovery-Algorithmus haben. Beim α-Algorithmus hängt das Ergebnis nur vom Log ab, und kann nicht über eine Gewichtung der Konformanzdimensionen gesteuert werden. Wir können lediglich im Nachhinein das Ergebnis bewerten. Bei neueren, insbesondere genetischen Algorithmen [17, 79] ist eine Gewichtung der Konformanzdimensionen Teil der Eingabe. Aus einer Reihe möglicher Ergebnisse wählt ein solcher Algorithmus dann eines der Prozessmodelle mit der besten Gesamtkonformanz.

In unserem Ansatz betrachten wir *Fitness* und *Einfachheit*. Für Präzision und Generalisierung ein Maß anzugeben wird schwierig bis unmöglich: Da wir Daten einführen, müssten wir für diese beiden Dimensionen Datenwerte in Betracht ziehen, die im Log nicht aufgezeichnet sind, aber theoretisch in einem gegeben Datentyp auftreten können. Auch wenn wir endliche Datendomänen wie Integer in Programmiersprachen annehmen, ist der Bereich so groß, dass wir sie zu den gesehenen Werten im Log kaum sinnvoll ins Verhältnis setzen können. Wenn wir theoretisch sogar von unendlichen Datendomänen wie den natürlichen Zahlen ausgehen, ist es nicht mehr möglich, die gesehenen Abläufe mit allen möglichen ins Verhältnis zu setzen.

8.3 Coloured Petri Nets

Wir führen nun kurz *gefärbte Petrinetze* (englisch: Coloured Petri Nets, CPN) ein, mit denen wir die Adapterpatterns modellieren. Wir betrachten die Struktur und geben eine Intuition für die Schaltregel an. Für eine ausführliche Besprechung von CPN sei auf Jensen und Kristensen [47] verwiesen.

CPN erweitern klassische Petrinetze um Datentypen, genannt *Farbmengen*, mit den entsprechenden Daten, genannt *Farben*. Daher sprechen wir auch von gefärbten Petrinetzen. Ein klassisches Petrinetz können wir als gefärbtes Petrinetz mit einer einelementigen Farbmenge interpretieren. Grundsätzlich besteht die Struktur eines CPN auch aus Plätzen, Transitionen und Kanten.

Bei einem CPN wird jedem Platz ein Typ zugeordnet. Das bedeutet, dass ein Platz eine Multimenge über einer Farbmenge beschreibt; anders ausgedrückt ein Platz hält gefärbte Marken vor. Zwei verschiedene Marken können dabei die gleiche Farbe repräsentieren.

Kanten in einem CPN sind grundsätzlich mit einem Label versehen. Entweder gibt das Label direkt die Farbe vor, die von einem Platz konsumiert oder auf einem Platz produziert werden soll, oder das Label ist eine Variable der dem Platz entsprechenden Farbmenge. An Kanten von einer Transition zu einem Platz können auch Funktionen stehen, deren Wertebereich der Farbmenge des Platzes entspricht.

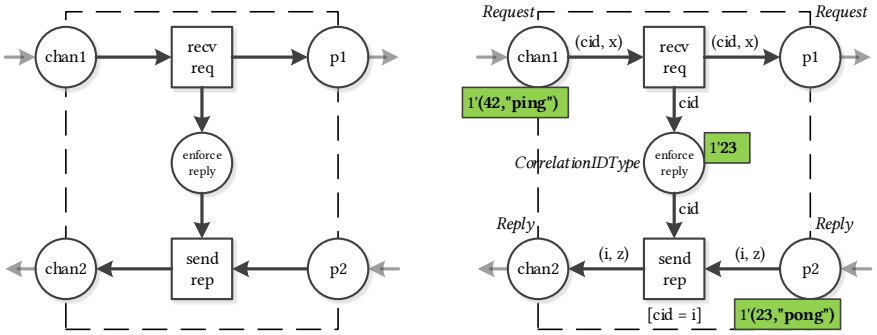
Mit Hilfe einer Bedingung können wir das Schalten einer Transition einschränken. Nur wenn die Bedingung wahr ist, darf die Transition schalten.

Ein Markierung ordnet nun einem Platz nicht nur eine Zahl von Marken zu, sondern eine Multimenge an gefärbten Marken.

Ein Beispiel für ein CPN, dessen Struktur und den Einsatz von Farben sehen wir in Abbildung 45. Das gezeigte Netz ist eine Implementierung eines Request-Reply-Patterns aus den Enterprise Integration Patterns, in dem oben eine Anfrage von links nach rechts geht und unten die entsprechende Antwort von rechts nach links. Die gestrichelte Linie dient der Abgrenzung des Patterns und sei für uns an dieser Stelle nicht von Interesse.

Auf der linken Seite in Abbildung 45a sehen wir die Struktur des Request-Reply-Patterns. Oben soll eine Anfrage weitergeleitet werden, und unten deren Antwort. Wir sehen die Plätze chan1, chan2, p1, p2 und enforce reply sowie die Transitionen recv req und send rep mit den entsprechenden Kanten dazwischen. Die Namen geben wir hier bei Plätzen und Transition im Knoten an.

Auf der rechten Seite in Abbildung 45b sehen wir zusätzlich noch Beschriftungen an Plätzen, Transitionen und Kanten. Ein Platz hat grundsätzlich einen Typ. Die Typen in der Abbildung sind *Request*, *Reply* und *CorrelationIDType*. Der Platz chan1 ist somit vom Typ *Request*. Die beiden Typen *Request* und *Reply* seien Datentypen, die jeweils



(a) Struktur eines CPN (b) Daten in einem CPN

Abbildung 45: Struktur und Daten/Farben in einem CPN

ein Tupel aus einer natürlichen Zahl und einer Zeichenkette beschreiben. Der Typ *CorrelationIDType* repräsentiert eine natürliche Zahl.

Die Beschriftung an den Kanten muss mit dem Typ des verbundenen Platzes übereinstimmen. So ist die Variable cid an der Kante zwischen dem Platz enforce reply und der Transition send rep vom Typ *CorrelationIDType*. Bei einem komplexen Datentyp benutzen wir Variablen für einen Teil des komplexen Datentyps entsprechend dem Typ des Teils. Die Beschriftung (cid, x) der Kante zwischen dem Platz chan1 und der Transition recv req bedeutet, dass von chan1 ein Tupel konsumiert wird, dessen erster Teil an die Variable cid und dessen zweiter Teil an die Variable x gebunden wird.

Eine Transition können wir mit einer Bedingung, also einem Ausdruck, der zu wahr oder falsch ausgewertet wird, beschriften. Dieser Ausdruck wird in eckige Klammern gesetzt. Im Beispiel betrifft dies die Transition send rep mit der Beschriftung $[\text{cid} = \text{i}]$. Die Transition schaltet also höchstens dann, wenn die beiden Variablen cid und i gleich belegt sind.

Die Markierung eines Platzes ist im zugehörigen grünen Kasten dargestellt. Die allgemeine Notation ist dabei: „Anzahl der Marken“ „Hochkomma“ „Datenwert“. Der Platz chan1 trägt somit 1 Marke mit dem Wert (42, "ping"), der Platz enforce reply 1 Marke mit dem Wert 23 und der Platz p2 1 Marke mit dem Wert (23, "pong").

Die Markierung eines CPN entscheidet, ob eine Transition aktiviert ist. Wie bei klassischen Petrinetzen muss auf jedem Vorplatz eine Marke liegen. Zusätzlich müssen die

Kantenbeschriftungen beachtet werden und eine eventuelle Bedingung der Transition muss erfüllt sein.

Steht an einer Kante ein konkreter Datenwert, dann muss auf dem entsprechenden Vorplatz eine Marke mit diesem Wert liegen. Steht an einer Kante ein Variable, dann kann diese Variable mit einem beliebigen Wert belegt werden, allerdings muss bei allen Vorplätzen, deren Kanten die gleiche Variable benutzen, der gleiche Wert zur Verfügung stehen.

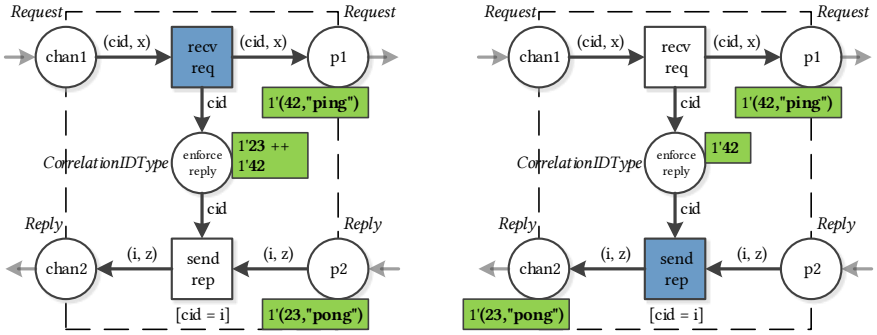
Betrachten wir im Beispiel in Abbildung 45b die Transition recv req. Diese hat genau einen Vorplatz chan1, die Kantenbeschriftung ist (cid, x). Da es keine Bedingung für recv req gibt, kann die Transition also genau dann schalten, wenn der Platz chan1 ein Marke besitzt, die ein Tupel ist. Die Variable cid wird dabei an den ersten Teil des Tupel gebunden und die Variable x an den zweiten Teil des Tupel. Mit der Belegung der Variablen cid mit 42 und x mit "ping" ist die Transition aktiviert.

Dieselbe Argumentation gilt für die Transition send rep und den Platz p2, allerdings gibt es hier noch den Platz enforce reply im Vorbereich. Von diesem konsumiert die Transition einen Wert, der an die Variable cid gebunden ist. Die Bedingung [cid = i] besagt, dass die Transition nur schalten darf, wenn an cid und i der gleiche Wert gebunden werden kann. Das gleiche Verhalten ohne Bedingung würden wir erreichen, wenn die Kante zwischen enforce reply und send rep mit i beschriftet wäre. Mit der Belegung der Variablen cid und i mit jeweils 23 und z mit "pong" ist die Transition aktiviert.

In Abbildung 46 sehen wir den Effekt des Schaltens der Transition recv req und anschließend der Transition send rep.

Beim Schalten der Transition recv req wird cid an den Wert 42 gebunden und x an den Wert "ping". In Abbildung 46a wurde die entsprechende Marke (42, "ping") von Platz chan1 konsumiert. Mit diesen Variablenbelegungen ergibt sich, dass auf Platz p1 eine Marke mit dem Wert (42, "ping") und auf Platz enforce reply eine Marke mit dem Wert 42 produziert wird. Die Notation ++ steht für eine Aufzählung von Marken; es liegen also die beiden Marken 23 und 42 auf dem Platz enforce reply.

Die Transition recv req ist nun nicht mehr aktiviert, weil keine Marke auf Platz chan1 liegt. Für die Transitionen send rep gibt es zwei mögliche Variablenbindung. Da auf Platz p2 jedoch nur die Marke (23, "pong") liegt, kann die Variable i nur mit 23 und z nur mit "pong" belegt werden. Da auf Platz enforce reply zwei Marken liegen, kann die Variable cid sowohl mit 23 als auch mit 42 belegt werden. Im zweiten Fall ist jedoch



(a) Schalten von recv req (b) Schalten von send rep

Abbildung 46: Effekt des Schaltens der Transitionen recv req und send rep ausgehend von Abbildung 45b

die Bedingung der Transition send rep nicht erfüllt. Somit ist in der einzig möglichen Belegung cid an 23 gebunden.

Den Effekt des Schaltens der Transition send rep für die besprochene Bindung der Variablen sehen wir in Abbildung 46b. Die Transition konsumiert die beiden Marken von den Plätzen p2 und enforce reply und produziert eine neue Marke (23, "pong") auf dem Platz chan2.

8.4 Definieren von Adapterkandidaten

Die Idee in diesem Kapitel ist es, aus gegebenen Patterns einen Kandidaten für einen Adapter zu erzeugen, und dann zu überprüfen, wie gut der Kandidat gegebenen Logs erklären kann. Bevor wir im nächsten Abschnitt sehen, wie wir die Güte bezüglich der Konformanz eines Adapterkandidaten bestimmen, betrachten wir zuerst die Erzeugung von Kandidaten aus Patterns.

Gegeben seien Logs mit ihren Kommunikationsereignissen und eine Menge von Patterns. Wir nehmen hier ausschließlich CPN-Patterns an. Dies schränkt uns jedoch nicht ein, da wir allgemein verbreitete Patterns wie die Enterprise Integration Patterns in CPN-Patterns übersetzen können.

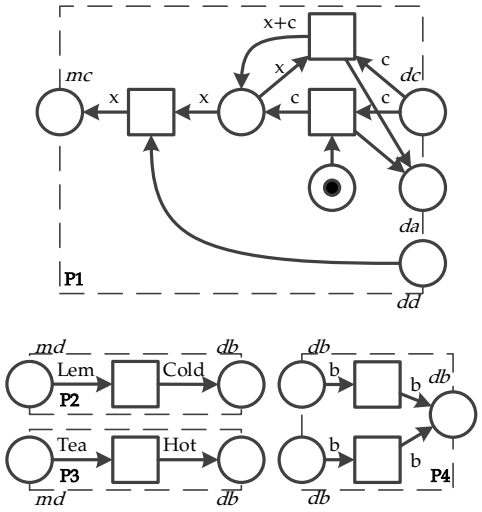


Abbildung 47: Vier CPN-Pattern P1, ..., P4 in unserem Beispiel

Da wir Patterns miteinander kombinieren wollen, definieren wir, dass ein CPN-Pattern eine Schnittstelle zu anderen Patterns hat. Da wir den Nachrichtenfluss zwischen und innerhalb der einzelnen Patterns modellieren, nehmen wir für jedes Pattern an, dass es eine ausgezeichnete Menge an Plätzen gibt, die als Schnittstelle dienen.

Definition 57 (CPN-Pattern)
Ein *CPN-Pattern N* ist ein CPN mit einer Menge P_{in} von Eingangsplätzen und einer Menge P_{out} von Ausgangsplätzen. Die Plätze $P_{in} \cup P_{out}$ bilden die Schnittstelle von N.

In Abbildung 47 sehen wir Beispiele solcher Patterns, wie wir sie bereits im einführenden Beispiel genutzt haben. Ein Pattern ist durch eine gestrichelte Linie abgegrenzt und hat einen Namen, wie zum Beispiel P1. Die Ein- und Ausgangsplätze liegen auf der gestrichelten Linie. Da die Namen der einzelnen Knoten in diesem Kapitel größtenteils nicht relevant, lassen wir sie in der graphischen Darstellung weg. Wir beschriften allerdings einen Schnittstellenplatz mit seinem Typ in kursiver Schrift.

Im Beispiel in Abbildung 47 sehen wir drei verschiedene Arten von Pattern. Die beiden Pattern P2 und P3 sind zwar mit unterschiedlichen Werten an den Kanten

beschriftet, aber sie sind strukturell gleich und ihr Schnittstellenplätze besitzen die gleichen Typen. Dass ein Pattern unterschiedlich ausgeprägt sein kann, ist fundamental für den patternbasierten Entwurf von Systemen.

Das Pattern P1 aggregiert verschiedene Münzbeträge c zu einem Gesamtbetrag x aggregieren. Die Eingangsplätze haben je den Typ *dc* beziehungsweise *dd*, die Ausgangsplätze haben den Typ *da* beziehungsweise *mc*.

Die beiden Patterns P2 und P3 übersetzen jeweils einen Typ in einen anderen. Das Pattern P2 übersetzt die Limonade Lem in ein Kaltgetränk Cold, und das Pattern P3 übersetzt den Tee Tea in ein Heißgetränk Hot. Beide Patterns haben jeweils einen Eingangsplatz vom Typ *md* und einen Ausgangsplatz vom Typ *db*.

Das letzte Pattern P4 dient der Weiterleitung einer Nachricht aus zwei Quellen an ein Ziel. Daher haben sowohl die beiden Eingangsplätze als auch der Ausgangsplatz den Typ *db*.

Wir komponieren CPN-Patterns, indem wir einen Eingangsplatz eines Patterns mit dem Ausgangsplatzes eines zweiten Patterns verschmelzen, wobei Ein- und Ausgangsplatz vom selben Typ sein müssen. So können Nachrichten eines bestimmten Typs von einem zum nächsten Pattern geschickt werden.

Definition 58 (Komponieren von CPN-Patterns)
Seien N_1 und N_2 zwei CPN-Patterns, sodass N_1 einen Ausgangsplatz vom Typ t und N_2 einen Eingangsplatz vom Typ t besitzt. Wir *komponieren* N_1 und N_2, indem wir den Ausgangsplatz von N_1 und den Eingangsplatz von N_2 verschmelzen, sodass der verschmolzene Platz p genau den Vorbereich des Ausgangsplatzes und genau den Nachbereich des Eingangsplatzes hat. Der Platz p ist dann nicht mehr Teil der Schnittstelle.

Jedes Pattern soll, wenn es benutzt wird, um einen Kandidaten zu erzeugen, vollständig benutzt werden. Es sollen also weder Ein- noch Ausgangsplätze unverbunden sein. Bei Modellierungspatterns hat jeder Teil der Schnittstelle eine besondere Bedeutung. Einen solchen Teil nicht zu nutzen widerspräche somit der Aufgabe eines bestimmten Patterns.

Bei der *vollständigen Komposition* der Patterns müssen wir jedoch beachten, dass wir einen Kandidaten für einen Adapter generieren wollen; wir benötigen also eine

Schnittstelle zu den System S_1 und S_2, welche die Logs erzeugt haben. Wir definieren dafür spezielle CPN-Log-Pattern.

Definition 59 (Log-Pattern)
Sei L ein Log. Dann bezeichnen wir ein CPN-Pattern N als *Log-Pattern* für L, wenn die Schnittstelle von N genau den Kommunikationsereignissen von L entspricht. Für jedes Empfangsereignis gibt es einen Eingangsplatz und für jedes Sendeereignis gibt es einen Ausgangsplatz.

Die beiden offenen Systeme in Abbildung 43a und Abbildung 43c sind somit Log-Patterns, da sie genau die Schnittstelle besitzen, die sich aus ihren entsprechenden Logs ergibt.

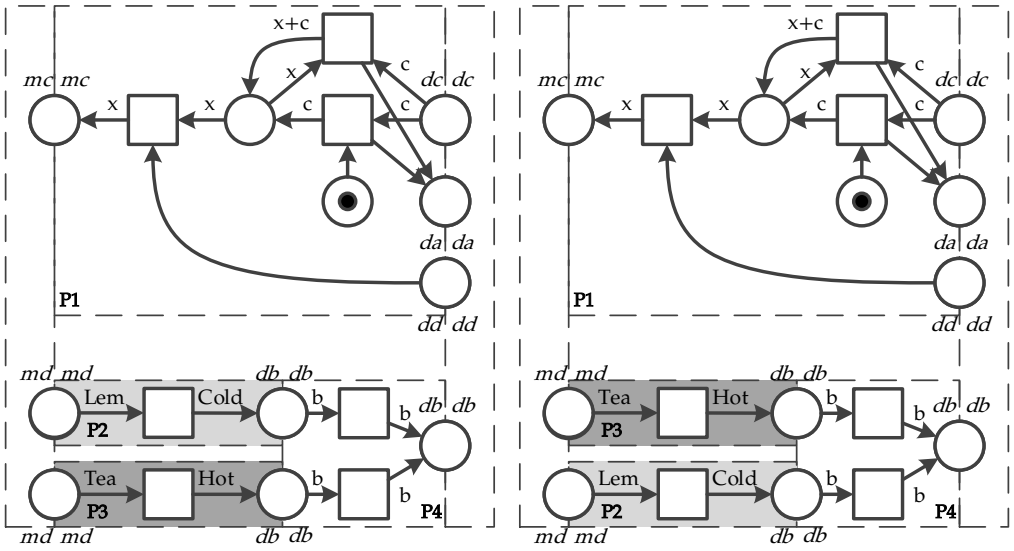
Mit den Log-Patterns können wir nun die Kandidaten definieren, für die wir im nächsten Abschnitt die Güte bestimmen wollen, um letztendlich einen Adapter zu finden, der das Kommunikationsverhalten der Logs bestmöglich erklärt.

Definition 60 (Adapterkandidat)
Seien L_1 und L_2 zwei Logs mit Kommunikationsereignissen, N_1 und N_2 deren entsprechende Log-Patterns, und $N_{P_1}, \ldots, N_{P_n}$ eine endliche Menge von CPN-Patterns. Wir bezeichnen eine Komposition der Patterns $N_{i_1}, \ldots, N_{i_k}$ ($\{i_1, \ldots, i_k\} \subseteq \{P_1, \ldots, P_n\}$) mit N_1 und N_2 als *Adapterkandidat*, wenn die Komposition vollständig ist.

Die Definition erlaubt, jedes CPN-Pattern höchstens einmal zu benutzen. Wenn wir Patterns mehrfach benutzen wollen, erweitern wir die Menge der CPN-Patterns einfach um entsprechende Kopien.

Wichtig für den Adapterkandidaten ist, wie er mit den Log-Patterns verbunden ist. Unterschiedliche Kommunikationsereignisse können den gleichen Typ besitzen. Daher gibt es auch unterschiedliche Kompositionsmöglichkeiten der CPN-Pattern mit den Log-Pattern. Das wiederum beeinflusst natürlich, welchen Einfluss ein Pattern auf das Verhalten der Komposition hat.

Mit dieser Definition erhalten wir eine endliche Menge an Adapterkandidaten $A_1, \ldots, A_l$. Da wir endlich viele Patterns betrachten, gibt es auch nur endlich viele Kombinationsmöglichkeiten der Ein- und Ausgangsplätze. Die Anzahl an Kandidaten kann jedoch exponentiell größer sein als die Anzahl der zur Verfügung stehenden Patterns. Im extremsten Fall haben alle Ein- und Ausgangsplätze den gleichen Typ.



(a) Kandidat 1 (b) Kandidat 2

Abbildung 48: Zwei Adapterkandidaten für das initiale Beispiel aus Abbildung 43, die Unterschiede sind farblich hervorgehoben.

Dann ist die Zahl der möglichen Kompositionen bei $2k$ Ein- und Ausgangsplätzen aufgrund der möglichen Kompositionen von je zwei solcher Plätze bei $k!$.

In Abbildung 48 sehen wir zwei Adapterkandidaten für unser initiales Beispiel. Wir benutzen jeweils alle vier Pattern P1 bis P4, am linken und rechten Rand der Abbildungen sehen wir jeweils Log-Pattern.

In Abbildung 48a benutzen wir die Patterns wie ursprünglich im Beispiel gegeben. Das Pattern P2 ist über dem Pattern P3 mit den gleich getypten Ein- und Ausgangsplätzen angeordnet. Im zweiten Adapterkandidaten in Abbildung 48b benutzen wir diese beiden Patterns in umgekehrter Reihenfolge. Zwei weitere Adapterkandidaten erhalten wir, indem wir von den beiden gezeigten Kandidaten ausgehen und die Patterns P2 und P3 über Kreuz mit dem Pattern P4 komponieren.

Strukturell und von den Typen her sind beide Kandidaten mögliche und sinnvolle Adapter. Allerdings müssen wir noch bewerten, welcher der beiden Kandidaten besser zu dem aufgezeichneten Verhalten passt. Aus dem einführenden Beispiel wissen wir,

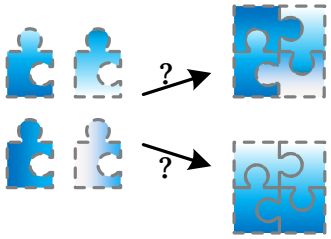


Abbildung 49: Patterns können unterschiedlich kombiniert werden, ergeben aber nicht immer ein sinnvolles Ergebnis.

dass der erste Kandidat genau der ist, der die beiden Systeme adaptiert. Beim zweiten Kandidaten werden die Getränke nicht vom Adapter entgegen genommen, da zwar die Typen stimmen, aber die Werte Tea und Lem nicht übereinstimmen. Über die Güte eines Adapterkandidaten erkennen wir dieses problematische Verhalten.

8.5 GÜTE EINES ADAPTERS

Wir wissen nun, wie wir aus Patterns mögliche Kandidaten konstruieren, die das aufgezeichnete Kommunikationsverhalten gegebener Logs erklären könnten. Wir müssen nun jedoch bestimmen, welcher Kandidat das Verhalten am besten erklärt. Die Situation ähnelt einem Puzzlespiel, bei dem wir die einzelnen Puzzleteile auf unterschiedlich zusammenlegen können, so wie in Abbildung 49 zu sehen.

Die Patterns bilden die Puzzleteile und ein Adapterkandidat bildet eine vollständige Instanz eines Puzzles. Wenn wir jedoch die Teile teilweise beliebig zusammenfügen dürfen, ist es fraglich, ob das Bild, das entsteht, sinnvoll ist. Im Fall der Adapterkandidaten betrifft das die Frage, ob das Verhalten eines Kandidaten die Logs gut erklären kann. Allein von der Struktur können wir nicht darauf schließen, sondern wir müssen die Kandidaten auswerten. Wir interessieren uns in diesem Zusammenhang dann für den Kandidaten, der das Verhalten der Logs am besten erklärt.

In Abschnitt 8.2 haben wir die vier typischen Konformanzdimensionen kennengelernt: Fitness, Präzision, Einfachheit und Generalisierung. Wir definieren in diesem Abschnitt, wie wir Fitness und Einfachheit eines Adapterkandidaten bestimmen, und warum mit unserer Methode Präzision und Generalisierung nicht anwendbar sind.

8.5.1 *Einfachheit*

Einfachheit ist die Dimension , die beschreibt, wie einfach ein Modell strukturiert ist. Dabei wird einfache Struktur in der Regel mit der Verständlichkeit eines Modells gleichgesetzt.

Ein Modell, das wenige Patterns benutzt, ist in der Regel leichter zu verstehen, als ein Modell, dass mehr Patterns benutzt. Im günstigsten Fall besteht ein Adapterkandidat aus einem einzigen Pattern; die Funktion des Adapterkandidaten entspricht dann der Funktion des Patterns. Ein naheliegendes Maß für die Einfachheit eines Adapterkandidaten besteht somit in der Anzahl der benutzten Patterns. Wir definieren das Einfachheitsmaß so, dass es größer ist, je einfach ein Adapterkandidat ist.

Definition 61 (Einfachheitsmaß I)
Seien L_1 und L_2 zwei Logs, $N_{P_1}, \ldots, N_{P_n}$ eine Menge von n CPN-Patterns und A ein Adapterkandidat aus diesen Patterns. Wir definieren das *Einfachheitsmaß* einfach_I wie folgt:

$$\text{einfach}_I(A) = 1 - \frac{\#(\text{benutzte Pattern in } A) - 1}{n}$$

Nach dieser Definition heißt ein Adapterkandidat einfacher, wenn er aus weniger Pattern besteht. Der einfachste Adapterkandidat, der nur aus einem Pattern besteht, hat das Einfachheitsmaß 1.

Auch wenn wir mit dem obigen Einfachheitsmaß Adapterkandidaten grob bezüglich ihrer Einfachheit einordnen können, gibt es doch Fälle, in denen dieses Maß zu ungenau ist. Wenn zwei Adapterkandidaten aus der gleichen Zahl an Pattern bestehen, kann es sein, dass ein Kandidaten nur aus sehr komplexen Patterns besteht, der andere dagegen aus sehr einfachen. Dies spiegelt sich im Fall der CPN-Patterns in der Anzahl der Plätze und Transitionen wieder.

Wir definieren, dass ein CPN-Pattern einfacher ist als ein anderes CPN-Pattern, wenn es weniger Knoten besitzt. In unserem Beispiel in 47 ist das Pattern P2 einfacher als das Pattern P4, und das Pattern P4 ist einfacher als das Pattern P1. Die beiden Patterns P2 und P3 sind diesbezüglich gleich einfach. Diese Definition erweitern wir auf einen Adapterkandidaten, indem wir die Anzahl der Knoten des Kandidaten betrachten.

Definition 62 (Einfachheitsmaß II)
Seien L_1 und L_2 zwei Logs, $N_{P_1}, \ldots, N_{P_n}$ eine Menge von n CPN-Pattern und A ein Adapterkandidat aus einem Teil dieser Patterns. Wir definieren das *Einfachheitsmaß* einfach_{II} wie folgt:

$$\text{einfach}_{II}(A) = 1 - \frac{\#(\text{Knoten in } A)}{\sum_{i=1,\ldots,n} \#(\text{Knoten in Pattern } N_{P_i})}$$

Mit diesem Maß ist ein Adapterkandidat unter Umständen einfacher, wenn er aus vielen kleinen Patterns anstelle weniger komplexer Patterns besteht.

Wir könnten an dieser Stelle noch weitere Maße definieren. So könnten wir zusätzlich die Flussrelation betrachten oder die Zahl der Verzweigungen in einen Adapterkandidaten. Letztlich ist es eine subjektive Entscheidung, was wir als einfach bezeichnen möchten.

8.5.2 *Fitness*

Die Dimension *Fitness* gibt an, inwieweit ein Prozessmodell aufgezeichnetes Verhalten in einem Log wiedergeben kann. Hier setzen wir die Zahl der Abläufe des Logs, die im Prozessmodell abgespielt werden können, mit allen Abläufen des Logs ins Verhältnis.

In unserem Fall ergibt sich jedoch eine Besonderheit: Wir können die beiden gegebenen Logs nicht individuell betrachten. Für eine bestimmte Instanz des ersten Systems, die einen Ablauf im Log verursacht hat, gibt es eine entsprechende Instanz des zweiten Systems. Da die Systeme Nachrichten austauschen, beeinflussen sie sich gegenseitig bezüglich der Ereignisse während eines Ablaufs, aber auch bezüglich der ausgetauschten Daten. In unserem Beispiel führt das zweifache Senden einer Münze durch den Kunden dazu, dass der Getränkeautomat Limonade zurückgibt. Der Adapter muss genau diesen Zusammenhang herstellen.

Wir setzen daher Paare von Abläufen in Relation. Wir wählen einen Ablauf aus dem ersten Log und einen Ablauf aus dem zweiten Log. Nun überprüfen wir, ob nicht nur die richtigen Ereignisse, sondern auch die richtigen Daten ausgetauscht werden. Wenn ein Paar von Abläufen komplett in einem Adapterkandidaten möglich ist, dann nennen wir das Paar bezüglich des Adapterkandidaten *korreliert*.

Replay eines Ablaufs

Um einen Ablauf tr eines Logs mit einem Adapterkandidaten abspielen zu können, definieren ein Log-Pattern N_{tr}. In diesem Log-Pattern treten die Ereignisse in der gleichen Reihenfolge auf wie in tr. Dies erreichen wir, indem wir die Sequenz der Ereignisse in tr in eine Sequenz von Transitionen übersetzen. Wir nennen dieses spezielle Log-Pattern ein *Replay-Pattern*.

Eine Transitionssequenz beginnt mit einen markierten Startplatz. Zwei Transitionen, die zwei aufeinander folgende Ereignisse darstellen, sind stets durch einen Kontrollflussplatz verbunden. Die Kante zwischen einer Transition und dem entsprechenden Ein- oder Ausgangsplatz ist mit dem zugehörigen Datenwert des Ereignisses beschriftet.

Die zwei Abläufe des Getränkeautomaten vom Anfang des Kapitels waren die beiden folgenden:

- $tr_1 = \langle ?m, time = 3, amount = 1\rangle, \langle !T, time = 6, bev = Tea\rangle$ und
- $tr_2 = \langle ?m, time = 7, amount = 2\rangle, \langle !L, time = 9, bev = Lemonade\rangle$.

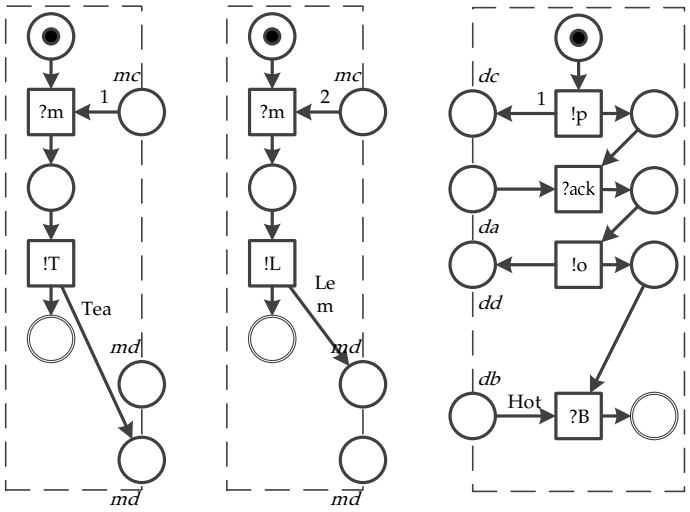
Für den Kunden nehmen wir folgenden Ablauf an:

- $tr_3 = \langle !p, time = 1, value = 1\rangle, \langle ?ack, time = 1\rangle, \langle !o, time = 2\rangle,$
  $\langle ?B, time = 11, drink = Hot\rangle$.

Die Umsetzung dieser drei Abläufe als Replay-Patterns sehen wir in Abbildung 50. Links ist der Ablauf tr_1 und in der Mitte der Ablauf tr_2 für den Getränkeautomaten. Rechts ist der Ablauf tr_3 des Kunden.

So sehen wir zum Beispiel im Fall von tr_1 in Abbildung 50a, dass zuerst die Transition ?m von einem Platz vom Typ mc den Datenwert 1 konsumiert. Dies entspricht dem ersten Teil $\langle ?m, time = 3, amount = 1\rangle$ des Ablaufs tr_1, wobei sich der Datenwert 1 auf den Wert von *amount* bezieht und wir vom Zeitpunkt abstrahieren. Wenn ?m geschaltet hat, der Getränkeautomat also die Münze erhalten hat, dann kann die Transition !T zum Senden des Tees feuern. Entsprechend dem zweiten Teil $\langle !T, time = 6, bev = Tea\rangle$ des Ablaufs tr_1 produziert die Transition !T den Datenwert Tea.

Für die Abläufe tr_2 und tr_3 ist die entsprechende Umsetzung in den Abbildungen 50b und 50c analog.



(a) tr_1 (b) tr_2 (c) tr_3

Abbildung 50: Drei verschiedene Abläufe als Log-Patterns

Ziel in allen drei Fällen ist es, dass der doppelt umrandete Platz am Ende markiert ist. Dies bedeutet, dass ein Ablauf komplett abgespielt werden konnte – sowohl bezüglich der Ereignisse als auch in Bezug auf die benutzen Datenwerte.

Mit Hilfe dieser Replay-Patterns können wir nun ein *Replay* definieren.

Definition 63 (Replay)
Seien L_1 und L_2 zwei Logs, $tr_1 \in L_1$ und $tr_2 \in L_2$ zwei Abläufe aus diesen Logs und N_1 und N_2 die dazugehörigen Replay-Pattern. Dann bezeichnen wir einen Adapterkandidaten in Komposition mit N_1 und N_2 als *Replay*.

Ein Replay ist *vollständig abspielbar*, wenn beide Replay-Pattern vollständig abgespielt werden können.

Durch ein Replay bestimmen wir, welche Abläufe in den Logs korreliert sind. Dazu müssen wir jedoch noch fordern, dass das Verhalten eines Replays *determiniert* ist. So stellen wir sicher, dass wir die vollständige Abspielbarkeit von Replays entschei-

den können. Im Allgemeinen ist die Erreichbarkeit einer bestimmten Markierung in gefärbten Petrinetzen unentscheidbar.

Definition 64 (Determiniertes Replay)
Wir nennen ein Replay *determiniert*, wenn in jedem Replay-Pattern N genau eine Markierung μ erreichbar ist, sodass μ kine Transition in N aktiviert.

Wenn ein Replay determiniert ist, erreichen wir an Ende jeder Ausführung des Replays die gleiche Markierung in einem Replay-Pattern. Wenn ein Replay also vollständig abspielbar ist, dann wird es immer vollständig abgespielt. Eine hinreichende Bedingung für dieses Verhalten ist die Abwesenheit von Konflikten zwischen aktivierten Transitionen.

Im Folgenden setzen wir voraus, dass alle Replays determiniert sind.

Definition 65 (Korrelierte Abläufe)
Seien L_1 und L_2 zwei Logs, und $tr_1 \in L_1$ und $tr_2 \in L_2$ zwei Abläufe. Wir nennen tr_1 und tr_2 bezüglich eines Adapterkandidaten *korreliert*, kurz $\langle tr_1, tr_2 \rangle$, wenn das Replay vollständig abspielbar ist.

Die Logs sind nicht vollkommen unabhängig entstanden, sodass für viele Abläufe im ersten Log ein entsprechender Ablauf in zweiten Log existieren sollte. Ein gutes Adaptermodell kann viele Abläufe korrelieren. Wir bilden daher Paare von korrelierten Abläufen und überprüfen, ob die Abläufe $tr_1 \in L_1$ und $tr_2 \in L_2$ miteinander korreliert sind. Dann betrachten wir keinen von den beiden Abläufen weiter, sondern überprüfen, in wieweit die restlichen Abläufe korreliert sind. So erhalten wir eine Zahl an Korrelationspaaren, die unser Fitnessmaß bildet.

Definition 66 (Fitnessmaß)
Seien L_1 und L_2 zwei Logs und A ein Adapterkandidat. Wir definieren das *Fitnessmaß* fit für A bezüglich L_1 und L_2 wie folgt:

$$\text{fit}(A) = \frac{\max(\#\text{Korrelationspaare})}{\min(\#\text{Abäufe in } L_1, \#\text{Abäufe in } L_2)}$$

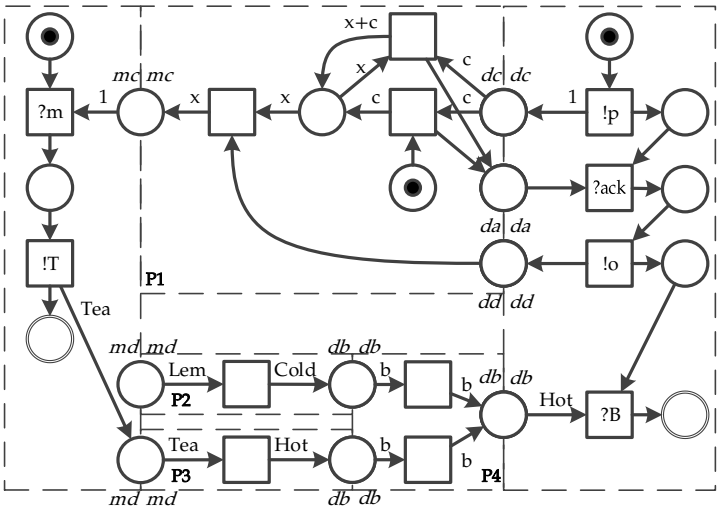


Abbildung 51: Replay aus einem Adapterkandidaten und den Abläufen tr_1 (links) und tr_3 (rechts)

Idealerweise können wir alle Abläufe eines Logs mit den Abläufen des anderen Logs korrelieren. Dies bedeutet perfekte Fitness für einen Adapterkandidaten. Zwei Beispiele zur Veranschaulichung des Fitnessmaß' sind in den Abbildungen 51 und 52 dargestellt.

Zuerst betrachten wir in Abbildung 51 das Replay eines Adapterkandidaten und den Abläufen tr_1 und tr_3. In Interaktion mit dem Pattern P1 können in tr_3 die Transitionen ǃp, ?ack und ǃo schalten. Aufgrund der letzten Transition produziert P1 eine Marke mit dem Datenwert 1 an der Schnittstelle zu tr_1. Auf der linken Seite schalten nun ?m und ǃT. Den Wert Tea kann sowohl Pattern P3 in Hot umwandeln als auch Pattern P4 weiterleiten, sodass letztendlich die Transition ?B in tr_3 schaltet. Damit wurden beide Abläufe vollständig abgespielt und die beiden Abläufe sind korreliert.

Das Replay des Adapterkandidaten und der Abläufe tr_2 und tr_3 in Abbildung 52 beginnt in tr_3 wie eben. Zuerst schalten die Transitionen ǃp, ?ack und ǃo. Damit legt das Pattern P1 eine Marke mit dem Datenwert 1 auf die Schnittstelle zu tr_2. Allerdings erwartet die Transition ?m für tr_2 den Datenwert 2 und kann somit nicht schalten. In dieser Situation kann keine weitere Transition schalten, das Replay ist nicht vollständig abspielbar und tr_2 und tr_3 sind somit nicht korreliert.

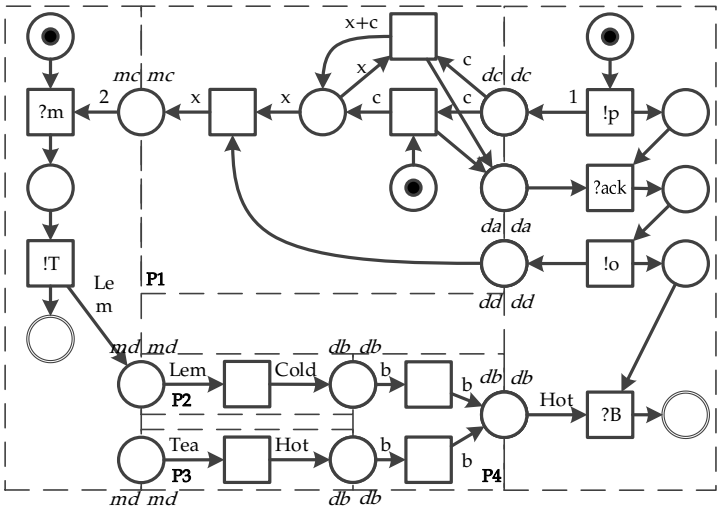


Abbildung 52: Replay eines Adapterkandidaten mit den Abläufen tr_2 (links) und tr_3 (rechts)

8.5.3 *Präzision und Generalisierung*

Die beiden Dimensionen *Präzision* und *Generalisierung* setzen auf geeignete Weise das mögliche Verhalten es Prozessmodells mit dem aufgezeichneten Verhalten in Beziehung. Für die Präzision interessiert uns, dass ein Prozessmodell möglichst wenig Verhalten über das aufgezeichnete Verhalten hinaus zeigen kann. Für die Generalisierung betrachten wir, welches mögliche Verhalten ein Prozessmodell zeigen kann, nachdem es teilweise abgespielt wurde.

In den von uns genutzten Logs und Pattern spielen Daten und Funktionen auf diesen Daten eine wesentliche Rolle. Damit muss die Frage, welche Datenwerte ausgetauscht werden können, Teil dieser beiden Dimensionen sein. Zum Beispiel ist ein Prozessmodell präziser, wenn es nur die aufgezeichneten Datenwerte betrachtet, als eines, das beliebige Datenwerte zulässt.

Allerdings sind die Definitionsbereiche gängiger Datentypen in der Regel sehr groß. Dies erlaubt es uns zwar theoretisch, alle möglichen Abläufe zu betrachten, allerdings ist deren Anzahl sehr viel größer im Verhältnis zum aufgezeichneten Verhalten, sodass wir kein vernünftiges Maß definieren können.

Eine mögliche Lösung, die zumindest teilweise erlauben würde, ein sinnvolles Maß für beide Dimensionen anzugeben, besteht in der Datenabstraktion. So könnten wir zum Beispiel Techniken wie abstrakte Interpretation benutzen, um einen Wertebereich in grobe Klassen zu unterteilen. Wir könnten auch ganz von Werten abstrahieren – das Netz quasi entfärben – und dann das mögliche Verhalten betrachten. Allerdings führt Abstraktion oft zu unzuverlässigen Ergebnissen.

Der Aufwand, ein sinnvolles Maß zu definieren, ist somit sehr hoch. Für den vorgestellten Ansatz, ein formales Adaptermodell abzuleiten, liegt der Fokus auf der Fitness. Ein Modell, dass nicht fit ist, kann auch nicht wirklich präzise sein, denn um möglichst nur das aufgezeichnet Verhalten erklären zu können, muss ein Modell überhaupt erst einmal das Verhalten erklären können. Wir verzichten daher im Rahmen dieser Arbeit auf die Angabe entsprechender Maße.

8.6 IMPLEMENTATION

Die Umsetzung des hier vorgestellten Ansatzes erfolgte im Werkzeug PETRA [37]. Dieses bekommt als Eingabe zwei Logs, eine Menge von CPN-Patterns und liefert den Adapterkandidaten mit der besten Konformanz.

8.6.1 *Verfügbare Bibliotheken*

Für die Speicherung von Logs im Process-Mining-Umfeld hat sich der XES-Standard [43] etabliert. Eine entsprechende Bibliothek erlaubt das Parsen, Manipulieren und Austauschen von Logs.

Aufgrund der Komplexität von Coloured Petri Nets und insbesondere der Schaltregel, greifen wir auf die Bibliothek ACCESS/CPN [113] zurück. Diese ermöglicht, Replays zu modellieren und anschließend auszuführen.

Um die beiden Bibliotheken nutzen zu können, erfolgte die Implementierung des Werkzeuges in Java.

8.6.2 *Strukturierung der Replays*

Bei der Umsetzung des Ansatzes ergaben sich Performanzprobleme, die zum einen der semantischen Komplexität der CPN entspringen, und zum anderen auf dem Aufbau der genutzten Bibliothek beruhen. Der ACCESS/CPN beigefügte Simulator zum Ausführen eins CPN muss starten, das CPN muss in ihn übertragen werden, er führt auf dem CPN eine Syntaxüberprüfung durch und muss die Beschriftungen auswerten, um aktivierte Transitionen zu identifizieren. Diese Schritte sind für jedes Replay, also jedes Paar von Abläufen aus den beiden Logs, von neuem nötig.

Um die Zahl der Simulatorstarts zu verringern, konstruieren wir ein CPN, dass es uns erlaubt, mehrere Replays gleichzeitig durchzuführen. Dieses CPN modelliert einzelne Ereignisse eines Ablaufs nicht als Transition, sondern als Wert in einem speziellen Ablauftyp. Dieser Ablauftyp ist eine Liste, in der jedes Element ein Ereignis repräsentiert. Beim Replay wird das erste Element einer solchen Liste entfernt und die dem Ereignis entsprechende Transition mit dem dazugehörigen Datenwert wird geschaltet. Ein Replay ist somit das Abarbeiten solcher Listen.

Da eine Liste eine spezielle Art von Marke ist, können wir initial alle Abläufe des Logs als Anfangsmarkierung im CPN hinterlegen. Zusätzliche Transitionen sorgen für die Wahl von Ablaufpaaren und werten das Ergebnis aus. Statt den Simulator für jedes Replay den Simulator neu zu starten, genügt ein einziger Aufruf. Der Nachteil davon ist, dass je mehr Marken vorhanden sind, die Überprüfung, ob eine Transition aktiviert ist, länger dauert.

8.6.3 *Ausnutzen weiterer Informationen*

Einen großen Einfluss auf die Laufzeit hat die Zahl der möglichen Ablaufpaare beider Logs. Schlimmstenfalls muss jeder Ablauf des ersten Logs mit jedem Ablauf zweiten Logs abgespielt werden. Die Anzahl der Replays ist dann quadratisch in der Anzahl der aufgezeichneten Abläufe.

Im Regelfall enthält ein Log zu jedem Ereignis auch einen Zeitstempel. Wenn wir die Zeitstempel des ersten und letzten Ereignisses eines Ablaufes betrachten, ergibt sich ein Zeitintervall, in der die zugehörige Instanz dieses Ablaufes aktiv war. Die andere Instanz, mit der die erste Instanz interagiert hat, muss etwa zu gleichen Zeit aktiv

gewesen sein. Daher können wir uns beim Replay auf Paare von Abläufen beschränken, deren Zeitintervalle sich zumindest teilweise überlappen.

In Experimenten zeigt diese Art der Vorauswahl eine spürbare Verbesserung bezüglich der Laufzeit. Wir konnten in vielen Fälle die Komplexität, die beim Betrachten aller Paare quadratisch bezüglich der Größe der Logs ist, vermeiden. Es ergab sich ein linearer Faktor in Bezug auf die Größe der Logs, der davon abhängt, wie viele Instanzen eines System parallel gelaufen sind. Die Anzahl dieser Instanzen ist in der Regel beschränkt.

Weitere Verbesserungen basieren auf statischer Analyse der Abläufe und der Adapterkandidaten. Wir könnten überprüfen, ob es strukturell überhaupt möglich ist, dass ein bestimmter Nachrichtentyp zu einem bestimmten Punkt im Ablauf empfangen werden kann.

In unserem Tool PETRA haben wir statische Analyse implementiert, die überprüft, ob zwei Abläufe korreliert sein könnten: Wenn in jedem der beiden Abläufe erstmalig eine Empfangsereignis aufgetreten ist, können wir dieses Paar als nicht korreliert ausschließen, wenn entweder die Empfangsereignisse im Ablauf nicht möglich sind, oder der falsche Datenwert empfangen wird. Typischerweise hängt ein Empfangsereignis und der empfangene Wert vom Adapterkandidaten und dem zweiten Ablauf ab. Wenn ein Adapterkandidat die beiden Abläufe nicht korreliert, ist es wahrscheinlich, dass bereits das erste Empfangsereignis nicht korrekt ist im Replay.

Wir untersuchen als vorbereitende Maßnahme, welche Ablaufpaare jeweils bis zum ersten Empfangsereignis keinen Fehler zeigen. Alle anderen können wir sicher als nicht korreliert ausschließen. Wenn ein Ablaufpaar keinen Fehler bis zum ersten Empfangsereignis zeigt, dann kann es immer noch sein, dass später ein Fehler beim Abspielen auftritt.

Wir könnten diesen Ansatz auch iterativ weiterführen, also bis zum zweiten Empfangsereignis prüfen, dann bis zum dritten, und so weiter. Allerdings hat sich in Experimenten gezeigt, dass wir zwar durch die Vorauswahl die Paare möglicher korrelierter Abläufe weiter einschränken, die Laufzeit jedoch mit jeder Iteration deutlich ansteigt.

8.7 Zusammenfassung

In diesem Kapitel haben wir betrachtet, inwiefern wir ein formales Modell eines Adapters erstellen können, wenn wir über keine formalen Modelle offener Systeme verfügen, sondern nur über deren aufgezeichnetes Verhalten im Form von Logs.

Wie Ereignisse, insbesondere ausgetauschte Nachrichten, in den Logs zusammenhängen dürfen, geben wir mit Patterns vor. Einen Kandidaten für einen Adapter erhalten wir, indem wir die Patterns sinnvoll kombinieren.

Wir definieren für Kandidaten die Konformanzdimensionen Einfachheit und Fitness. Diese erlauben es uns, verschiedene Kandidaten zu bewerten und einen Kandidaten zu bestimmen, der das Verhalten der adaptierten Systeme am besten erklärt.

Die Patterns, die wir benutzen, konzentrieren sich auf den Datenfluss. Durch die Betrachtung von Datenwerten können wir in vielen Fällen Nichtdeterminismus im Verhalten, der durch die Abstraktion von Daten auftritt, auflösen. In zukünftiger Arbeit kann der Ansatz jedoch so erweitert werden, dass nicht nur der Datenfluss, sondern auch der Kontrollfluss abgeleitet wird.

Eine alternative Idee besteht darin, eine Engine zu erzeugen, die eine vorgegeben Menge an Transformationsregeln benutzt. Die Aufgabe in der Process Discovery wäre dann, einen Controller für den Adapter abzuleiten. Dies ist jedoch mit der bestehenden Technik nicht möglich, da die Controllersynthese die formalen Modelle der offenen Systeme benötigt, die jedoch nicht gegeben sind. Zuerst formale Modelle für die offenen Systeme abzuleiten, lässt uns zum einen keine Möglichkeit, Einfluss auf die Güte des Adaptermodells zu nehmen, zum anderen ist es möglich, dass die formalen Modelle verhaltensinkompatibel sind, sodass wir keinen Adapter synthetisieren können.

Sinnvoller ist es daher, den Controller bezüglich der Logs zu erstellen, auch wenn dies nicht mit den bekannten Algorithmen möglich ist. Müller u. a. [79] definieren für offene Netze einen genetischen Algorithmus, auf dessen Basis wir einen Controller ableiten können. Allerdings startet dieser Algorithmus auf Basis des allgemeinsten Kommunikationspartners, den wir nicht kennen. Wir müssen daher einen Partner definieren, der uns eine ähnliche Ausgangssituation für die Anwendung dieses Algorithmus bietet. Dies ist jedoch Teil zukünftiger Forschung.

Ergebnisse aus dem Gebiet der *Correlation Discovery* [7, 42, 77] ermöglichen es zumindest, die Abläufe der beiden betrachteten Logs in Relation zu setzen und so die Anzahl der Replays gering zu halten. Allerdings ergibt sich hier die Herausforderung

mögliche Datentransformationen in die Techniken mit einzubeziehen, die bewirken, dass Daten nicht 1-zu-1 zwischen Logs korreliert werden können, sondern bezüglich der entsprechenden Transition.

Teil III

ZUSAMMENFASSUNG UND AUSBLICK

9 ZUSAMMENFASSUNG UND AUSBLICK

9.1 ZUSAMMENFASSUNG DER RESULTATE

In dieser Arbeit haben wir die Adaption offener Systeme betrachtet. Für zwei offene Systeme S_1 und S_2 generieren wir einen Adapter A, sodass die Komposition $S_1 \oplus A \oplus S_2$ korrekt bezüglich der Semantik der ausgetauschten Nachrichten und des Verhaltens ist.

Kapitel 3

Die Technik zur Synthese, die wir hier präsentiert haben, betrachtet den Datenfluss und den Kontrollfluss getrennt voneinander. Um den Datenfluss zu modellieren, benutzen wir Transformationsregeln, die beschreiben, wie Nachrichten ineinander umgewandelt werden können. Methoden wie Schema Matching, semantische Analyse oder auch der musterbasierte Entwurf erlauben uns, die Transformationsregeln auf unterschiedlichste Art abzuleiten. Wir führen die Engine als den Teil eines Adapters ein, der die Transformationsregeln umsetzt und somit semantische Kompatibilität zwischen den offenen Systemen herstellt. Den Kontrollfluss generieren wir automatisch, indem wir auf Methoden der Controllersynthese zurückgreifen. Der so generierte Controller stellt sicher, dass der Adapter, der aus Engine und Controller besteht, verhaltenskompatibel mit den offenen Systemen interagiert.

Kapitel 4

Wir konnten zeigen, dass die Existenz eines Adapters nur von der Semantik der Nachrichten abhängt. Um einen Adapter generieren zu können, müssen die Transformationsregeln sinnvoll gewählt sein. Bei der Anwendung der Technik auf Firewallsysteme konnten wir ausnutzen, dass wir mit den Transformationsregeln leicht den Datenfluss beschreiben und die Engine auf das gewünschte Kommunikationsmodell anpassen

können. Entsprechend lässt sich die Technik auf weitere kommunizierende Systeme übertragen und auf die Anforderungen an den Datenfluss und den Kontrollfluss anpassen. Um aufzuzeigen, welche Möglichkeiten wir haben, die Synthese des Kontrollflusses zu beeinflussen, sind wir auf existierende Arbeiten zu Verhaltenseinschränkungen eingegangen. So können wir ohne Änderung des Korrektheitskriterium bestimmtes Verhalten im Adapter oder den offenen Systemen erzwingen oder vermeiden.

FESTSTELLUNG Aufgrund der konzeptionellen und technischen Trennung sind wir flexibel in der Beschreibung des Datenflusses und Kontrollflusses.

Im zweiten Teil der Arbeit beschäftigte uns die Frage, wie wir die Trennung in Datenfluss und Kontrollfluss ausnutzen können, um relevante Fragestellungen im Zusammenhang mit Adaptern zu beantworten.

Kapitel 6

Wenn wir für den Datenfluss zusätzlich vorgeben, wie dieser auf unterschiedliche Komponenten verteilt werden soll, können wir für den Kontrollfluss automatisch eine maximale Verteilung berechnen. Wir sind jedoch dadurch eingeschränkt, dass es Fälle gibt, in denen die vorgegebene Verteilung des Datenflusses keine Verteilung des Kontrollflusses erlaubt, dass sich die verteilten Komponenten genauso verhalten, wie der nichtverteilte Adapter.

Kapitel 7

Für den Fall, dass die Adaptersynthese keinen Adapter liefert, können wir auf die Erkenntnis zurückgreifen, dass die Existenz eines Adapters von den Transformationsregeln abhängt. Wir bestimmen die Punkte im Verhalten einer gegebenen Engine, an denen das Hinzufügen weiterer Regeln der Engine erlaubt, mehr Kommunikationsverhalten mit den gegebenen offenen Systemen zu zeigen. Wir diagnostizieren so, warum die Adaptersynthese fehlgeschlagen ist und zeigen Möglichkeiten auf, die Menge der Transformationsregeln sinnvoll zu erweitern.

Kapitel 8

Zuletzt haben wir betrachtet, wie wir ein formales Modell eines Adapters ableiten können, wenn uns keine formalen Modelle der offenen Systeme zu Verfügung stehen, sondern nur deren aufgezeichnetes Verhalten. Wir konzentrieren uns darauf, erst einmal den Datenfluss abzuleiten, sodass das abgeleitete Modell des Adapters das aufgezeichnete Verhalten der offenen Systeme gut erklärt. Wir benutzen Patterns, um die Möglichkeiten eines Adapters vorzugeben. Wir definieren verschiedene Konformanzmaße, die es uns erlauben, die Güte eines Adapters einzuschätzen. Die möglichen Adapter in Bezug auf die gegebenen Patterns ermitteln wir durch die vollständige Bestimmung aller Möglichkeiten, die Patterns zu kombinieren.

FESTSTELLUNG Aufgrund der technischen Trennung des Adapters in Datenfluss und Kontrollfluss sind wir in der Lage, Fragen bezüglich eines Adapters zielgerichtet zu beantworten.

Werkzeuge

Im Rahmen dieser Arbeit sind verschiedenste Werkzeuge entstanden. Jede Fragestellung der Arbeit, die wir konstruktiv oder algorithmisch beantworten, wurde entsprechend implementiert.

Das Werkzeuge MARLENE setzt die Technik zu Adaptersynthese um und kann Diagnoseinformationen bereitstellen. PETRA leitet formale Modelle für Adapter basierend auf Patterns ab.

Im Zusammenhang studentischer Arbeiten ist das Werkzeug CHARLOTTE entstanden, das einen synthetisierten Adapter ausführt und so Geschäftsprozesse adaptiert. Eine Bibliothek zur Manipulation offener Netze wurde so erweitert, dass sich offene Netze auf verschiedene Komponenten verteilen lassen.

Die Werkzeuge stehen quelloffen und frei auf service-technology.org zur Verfügung.

9.2 WEITERE FRAGESTELLUNGEN

Die in dieser Arbeit betrachteten Fragen schöpfen weder das Thema Adaptersynthese noch die Technik vollkommen aus. Wir stellen hier zwei weitere interessante Probleme

vor, wie wir das Ergebnis der Adaptersynthese weiter beeinflussen beziehungsweise die Trennung von Datenfluss und Kontrollfluss weiter ausnutzen können.

9.2.1 *Adaption bezüglich Austauschbarkeit*

Dumas, Spork und Wang [28] betrachten eine etwas andere Art von Adaptierung als in dieser Arbeit vorgestellt. Sie möchten, dass ein offenes System mit einer bestimmten Schnittstelle und einem bestimmten Verhalten nach außen hin eine andere Schnittstelle und anderes Verhalten zeigen. Sie haben also ein offenes System S_1 gegeben und möchten dieses so adaptieren, dass es wie ein offenes System S_2 aussieht. Die Autoren geben eine visuelle Notation an, um den benötigten Adapter A zu beschreiben, sodass die Äquivalenz $S_2 \equiv S_1 \oplus A$ gilt.

Diese Fragestellung lässt sich auf unsere Technik übertragen: Gegeben sei ein offenes Netz N_1 als gegebenes System und ein offenes Netz N_2 als gewünschtes System. Zudem gibt eine Menge von Transformationsregeln vor, wie die Nachrichten von N_1 mit N_2 im Zusammenhang stehen. Die Definition der Engine passen wir so an, dass sie auf der einen Seite mit N_1 komponiert wird und auf der anderen Seite die Schnittstelle von N_2 besitzt. Nun ist die Aufgabe, einen Controller C zu synthetisieren, sodass die Äquivalenz $N_1 \oplus C \oplus E \equiv N_2$ gilt, d. h., N_1 zusammen mit dem Adapter ist bezüglich seiner Kommunikation nicht von N_2 zu unterscheiden.

Die Frage, die wir hier eigentlich stellen, ist, ob es ein C gibt, sodass N_2 durch $N_1 \oplus C \oplus E$ *ausgetauscht* werden kann. Zur Austauschbarkeit offener Netze existieren bereits verschiedene Vorarbeiten [75, 101], deren Ergebnisse sich wahrscheinlich gut übertragen lassen. Somit wären wir in der Lage, eine etwas anders gelagerte Art der Adaption offener Systeme mit der vorgestellten Technik zu lösen.

9.2.2 *Fast korrekte Adapter*

Wir haben in dieser Arbeit Korrektheit absolut betrachtet. Der generierte Controller stellt sicher, dass der Adapter nur korrektes Verhalten in Interaktion mit den gegebenen offenen Systemen zeigt. Nun kann es sein, dass eines der offenen Systeme mit sehr geringer Wahrscheinlichkeit aufgrund externer Störeinflüsse Fehler zeigt. Allein die Möglichkeit, dass ein solcher Fehler auftritt, kann verhindern, dass die Adaptersynthese

erfolgreich ist. Auf den Fehler des offenen Systems kann unter Umständen kein zweites System angemessen reagieren; wir wollen den Fall aber in Kauf nehmen, wenn er nur selten auftritt.

Wir können uns alternativ fragen, ob es nicht einen Adapter gibt, der diesen Fehler zwar nicht beheben kann, aber zum Beispiel in 90 Prozent aller Abläufe korrektes Verhalten sicherstellt? Wenn das Auftreten des Fehlers bekannt ist und in Kauf genommen wird, nehmen wir auch für den Adapter an, dass der Fehler auftreten darf.

Eine wichtige Information, die wir für die Beantwortung dieser Art Fragen benötigen, ist die Wahrscheinlichkeit, mit der bestimmtes Verhalten auftreten kann. Wir benötigen zur Beschreibung eines offenen Systems stochastische Annotationen, um bestimmen zu können, mit welcher Wahrscheinlichkeit eine Aktion eines Ablaufes eintritt. *Stochastische Petrinetze* [71] bieten genau diese Möglichkeit.

In Zusammenhang mit der oben genannten Frage, müssen wir stochastische Petrinetze im Zusammenhang offener Systeme betrachten. Für die Controllersynthese müssen wir festlegen, welchen Einfluss ein Controller, aber auch die Engine auf die Wahrscheinlichkeitsverteilung hat.

DANKSAGUNG

Mein Dank gilt zuerst Wolfgang Reisig, der mir die Möglichkeit eröffnete, im Projekt „Synthese von Verhaltensadaptern“ (RE 834/18-1) der Deutschen Forschungsgemeinschaft zu arbeiten, das die Grundlage dieser Arbeit bildet. Besonderer Dank gilt auch Karsten Wolf und Dirk Fahland, die mir viele hilfreiche Impulse gaben und meine Ideen kritisch hinterfragt haben.

Arjan Mooij hat mir mit Beweistechniken und vielen subtilen Beispielen geholfen, die Möglichkeiten und Grenzen der Technik zur Adaptersynthese zu verstehen. Niels Lohmann hat durch seine Arbeit maßgeblich dazu beigetragen, dass sich die Tools einfach umsetzen und miteinander nutzen lassen. Christian Stahl ist eine wandelnde Bibliothek und konnte mir immer wieder nützliche Literaturhinweise geben.

Die letzten sechs Jahre hatte ich die Gelegenheit, mit wunderbaren Kollegen in Berlin zusammenzuarbeiten. Jan Sürmeli bot mir jederzeit ein offenes Ohr für fachliche Diskussionen. Andre Moelle hat sich viel Zeit genommen, um diese Arbeit zu lesen. Mit Robert Prüfer, Richard Müller, Raffael Dzikowski und Kim Völlinger entstanden immer wieder interessante Gespräche.

In diesem Zusammenhang gilt ein ganz besonders herzlicher Dank Birgit Heene, die mir jederzeit bei organisatorische Problemen geholfen hat und auch sonst immer ein hilfreicher Ansprechpartner war.

Ich habe die Gastfreundschaft in der Gruppe von Wil van der Aalst sehr genossen. Elham Ramezani, Joos Buijs und Dennis Schunselaar haben mich fern der Heimat freundlich in ihren Reihen aufgenommen. Dank Ine van der Ligt war es immer einfach und angenehm, wieder nach Eindhoven zu kommen.

Zu guter Letzt möchte ich mich bei meinen Eltern und meiner Freundin Ela bedanken, deren Unterstützung ich mir jederzeit sicher sein konnte.

A

LITERATURVERZEICHNIS

[1] W. M. P. van der Aalst, E. Kindler und J. Desel. „Beyond Asymmetric Choice: A note on some extensions“ In: *Petri Net Newsletter* 55 (1998), S. 3–13 (siehe S. 13).

[2] Wil M. P. van der Aalst. *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011, S. I–XVI, 1–352. ISBN: 978-3-642-19344-6 (siehe S. 14, 154, 157).

[3] Wil M. P. van der Aalst. „Service Mining: Using Process Mining to Discover, Check, and Improve Service Behavior“. In: *IEEE T. Services Computing* 6.4 (2013), S. 525–535. DOI: 10.1109/TSC.2012.25. URL: http://doi.ieeecomputersociety.org/10.1109/TSC.2012.25 (siehe S. 14).

[4] Wil M. P. van der Aalst. „The Application of Petri Nets to Workflow Management“. In: *Journal of Circuits, Systems, and Computers* 8.1 (1998), S. 21–66 (siehe S. 35, 98).

[5] *Apache ODE*. URL: http://ode.apache.org/ (besucht am 03. 07. 2014) (siehe S. 107, 111).

[6] Marco Autili u. a. „Synthesis of decentralized and concurrent adaptors for correctly assembling distributed component-based systems“. In: *Journal of Systems and Software* 81.12 (2008), S. 2210–2236. DOI: 10.1016/j.jss.2008.04.006. URL: http://dx.doi.org/10.1016/j.jss.2008.04.006 (siehe S. 13).

[7] Sujoy Basu, Fabio Casati und Florian Daniel. „Toward Web Service Dependency Discovery for SOA Management“. In: *2008 IEEE International Conference on Services Computing (SCC 2008), 8-11 July 2008, Honolulu, Hawaii, USA*. IEEE Computer Society, 2008, S. 422–429. DOI: 10.1109/SCC.2008.45. URL: http://doi.ieeecomputersociety.org/10.1109/SCC.2008.45 (siehe S. 179).

[8] Sandrine Beauche und Pascal Poizat. „Automated Service Composition with Adaptive Planning“. In: *Service-Oriented Computing - ICSOC 2008, 6th International Conference, Sydney, Australia, December 1-5, 2008. Proceedings*. Hrsg. von Athman Bouguettaya, Ingolf Krüger und Tiziana Margaria. 2008, S. 530–537. DOI: 10.1007/978-3-540-89652-4_42. URL: http://dx.doi.org/10.1007/978-3-540-89652-4_42 (siehe S. 11).

[9] Werner Van Belle, Tom Mens und Theo D'Hondt. „Using Genetic Programming to Generate Protocol Adaptors for Interprocess Communication“. In: *Evolvable Systems: From Biology to Hardware, 5th International Conference, ICES 2003, Trondheim, Norway, March 17-20, 2003, Proceedings*. Hrsg. von Andrew M. Tyrrell, Pauline C. Haddow und Jim Torresen. Springer, 2003, S. 422–433. DOI: 10.1007/3-540-36553-2_38. URL: http://dx.doi.org/10.1007/3-540-36553-2_38 (siehe S. 9).

[10] Boualem Benatallah u. a. „Developing Adapters for Web Services Integration“. In: *CAiSE*. 2005, S. 415–429 (siehe S. 8, 10, 52).

[11] Tim Berners-Lee und Mark Fischetti. *Weaving the web - the original design and ultimate destiny of the World Wide Web by its inventor*. HarperBusiness, 2000. ISBN: 978-0-06-251587-2 (siehe S. 7).

[12] Eike Best. „Structure Theory of Petri Nets: the Free Choice Hiatus“. In: *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course, Bad Honnef, 8.-19. September 1986*. Hrsg. von Wilfried Brauer, Wolfgang Reisig und Grzegorz Rozenberg. Springer, 1986, S. 168–205. DOI: 10.1007/BFb0046840. URL: http://dx.doi.org/10.1007/BFb0046840 (siehe S. 13).

[13] Andrea Bracciali, Antonio Brogi und Carlos Canal. „A formal approach to component adaptation“. In: *Journal of Systems and Software* 74.1 (2005), S. 45–54 (siehe S. 10, 52).

[14] Antonio Brogi, Carlos Canal und Ernesto Pimentel. „On the semantics of software adaptation“. In: *Sci. Comput. Program.* 61.2 (2006), S. 136–151. DOI: 10.1016/j.scico.2005.10.009. URL: http://dx.doi.org/10.1016/j.scico.2005.10.009 (siehe S. 10).

[15] Antonio Brogi und Razvan Popescu. „Automated Generation of BPEL Adapters“. In: *ICSOC*. 2006, S. 27–39 (siehe S. 12, 52, 108).

[16] Antonio Brogi u. a. „Formalizing Web Service Choreographies“. In: *Electr. Notes Theor. Comput. Sci.* 105 (2004), S. 73–94 (siehe S. 10, 52).

[17] Joos C. A. M. Buijs, Boudewijn F. van Dongen und Wil M. P. van der Aalst. „A genetic algorithm for discovering process trees“. In: *IEEE Congress on Evolutionary Computation*. 2012, S. 1–8 (siehe S. 159).

[18] Joos C. A. M. Buijs, Boudewijn F. van Dongen und Wil M. P. van der Aalst. „Quality Dimensions in Process Discovery: The Importance of Fitness, Precision, Generalization and Simplicity“. In: *Int. J. Cooperative Inf. Syst.* 23.1 (2014) (siehe S. 158).

[19] Diego Calvanese u. a. „Automatic Service Composition and Synthesis: the Roman Model“. In: *IEEE Data Eng. Bull.* 31.3 (2008), S. 18–22. URL: http://sites.computer.org/debull/A08Sept/roman.pdf (siehe S. 15).

[20] Javier Cámara u. a. „ITACA: An integrated toolbox for the automatic composition and adaptation of Web services“. In: *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 2009, S. 627–630. DOI: 10.1109/ICSE.2009.5070572. URL: http://dx.doi.org/10.1109/ICSE.2009.5070572 (siehe S. 12).

[21] Carlos Canal, Pascal Poizat und Gwen Salaün. „Model-Based Adaptation of Behavioral Mismatching Components“. In: *IEEE Trans. Software Eng.* 34.4 (2008), S. 546–563. DOI: 10.1109/TSE.2008.31. URL: http://doi.ieeecomputersociety.org/10.1109/TSE.2008.31 (siehe S. 11).

[22] Jing Cao und Albert Nymeyer. „A design methodology for verified web-service mediators“. In: *Computing* 95.7 (2013), S. 567–610. DOI: 10.1007/s00607-012-0237-5. URL: http://dx.doi.org/10.1007/s00607-012-0237-5 (siehe S. 11).

[23] Josep Carmona, Jordi Cortadella und Michael Kishinevsky. „New Region-Based Algorithms for Deriving Bounded Petri Nets“. In: *IEEE Trans. Computers* 59.3 (2010), S. 371–384 (siehe S. 46).

[24] Luca Cavallaro u. a. „Synthesizing adapters for conversational web-services from their WSDL interface“. In: *2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2010, Cape Town, South Africa, May 3-4, 2010*. Hrsg. von Rogério de Lemos und Mauro Pezzè. ACM, 2010, S. 104–113. DOI: 10.1145/1808984.1808996. URL: http://doi.acm.org/10.1145/1808984.1808996 (siehe S. 7).

[25] Erik Christensen u. a. *Web Services Description Language (WSDL)*. Hrsg. von W3C. 15. März 2001. URL: http://www.w3.org/TR/wsdl (besucht am 08. 07. 2014) (siehe S. 7, 112).

[26] Edmund M. Clarke, Orna Grumberg und Doron Peled. *Model checking*. MIT Press, 2001. ISBN: 978-0-262-03270-4 (siehe S. 3).

[27] Aurèle Destailleur. „Verteilung von Service-Adaptern“. Studienarbeit. Humboldt-Universität zu Berlin, Jan. 2013 (siehe S. 137).

[28] Marlon Dumas, Murray Spork und Kenneth Wang. „Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation“. In: *Business Process Management*. 2006, S. 65–80 (siehe S. 10, 52, 55, 186).

[29] Marlon Dumas u. a. „Understanding Business Process Models: The Costs and Benefits of Structuredness“. In: *CAiSE*. 2012, S. 31–46 (siehe S. 158).

[30] Islam Elgedawy. „Automatic generation for web services conversations adapters“. In: *The 24th International Symposium on Computer and Information Sciences, ISCIS 2009, 14-16 September 2009, North Cyprus*. IEEE, 2009, S. 616–621. DOI: 10.1109/ISCIS.2009.5291892. URL: http://dx.doi.org/10.1109/ISCIS.2009.5291892 (siehe S. 8).

[31] Dirk Fahland und Christian Gierds. „Analyzing and Completing Middleware Designs for Enterprise Integration Using Coloured Petri Nets“. In: *CAiSE*. 2013, S. 400–416. DOI: 10.1007/978-3-642-38709-8_26 (siehe S. 5, 52, 154).

[32] Kathrin Figl, Jan Recker und Jan Mendling. „A study on the effects of routing symbol design on process model comprehension“. In: *Decision Support Systems* 54.2 (2013), S. 1104–1118 (siehe S. 158).

[33] Matthew Fuchs. „Adapting Web Services in a Heterogeneous Environment“. In: *Proceedings of the IEEE International Conference on Web Services (ICWS'04), June 6-9, 2004, San Diego, California, USA*. IEEE Computer Society, 2004, S. 656. DOI: 10.1109/ICWS.2004.20. URL: http://doi.ieeecomputersociety.org/10.1109/ICWS.2004.20 (siehe S. 7).

[34] Erich Gamma u. a. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994 (siehe S. 6).

[35] Christian Gierds. „Finding Cost-Efficient Adapters“. In: *AWPN*. 2008, S. 37–42 (siehe S. 101, 104).

[36] Christian Gierds. *Marlene*. 2014. URL: http://service-technology.org/marlene/ (besucht am 07. 07. 2014) (siehe S. 109, 150).

[37] Christian Gierds. *Petra*. 2013. URL: http://service-technology.org/marlene/ (besucht am 05. 08. 2014) (siehe S. 176).

[38] Christian Gierds und Dirk Fahland. „Discovering Pattern-Based Mediator Services from Communication Logs“. In: *WESOA*. 2013 (siehe S. 154).

[39] Christian Gierds, Arjan J. Mooij und Karsten Wolf. „Reducing Adapter Synthesis to Controller Synthesis“. In: *IEEE T. Services Computing* 5.1 (2012), S. 72–85. DOI: 10.1109/TSC.2010.57 (siehe S. 51).

[40] Rob J. van Glabbeek, Ursula Goltz und Jens-Wolfhard Schicke. „On Synchronous and Asynchronous Interaction in Distributed Systems“. In: *CoRR* abs/0901.0048 (2009) (siehe S. 13, 16, 119, 120, 125, 137).

[41] Robert Gombotz und Schahram Dustdar. „On Web Services Workflow Mining“. In: *Business Process Management Workshops, BPM 2005 International Workshops, BPI, BPD, ENEI, BPRM, WSCOBPM, BPS, Nancy, France, September 5, 2005, Revised Selected Papers*. Hrsg. von Christoph Bussler und Armin Haller. 2005, S. 216–228. DOI: 10.1007/11678564_19. URL: http://dx.doi.org/10.1007/11678564_19 (siehe S. 14).

[42] Adnene Guabtni, Hamid R. Motahari Nezhad und Boualem Benatallah. „Using Graph Aggregation for Service Interaction Message Correlation“. In: *Advanced Information Systems Engineering - 23rd International Conference, CAiSE 2011, London, UK, June 20-24, 2011. Proceedings*. Hrsg. von Haralambos Mouratidis und

Colette Rolland. Springer, 2011, S. 642–656. DOI: 10.1007/978-3-642-21640-4_47. URL: http://dx.doi.org/10.1007/978-3-642-21640-4_47 (siehe S. 14, 179).

[43] Christian W. Günther und Eric Verbeek. *XES Standard Definition*. 28. März 2014. URL: http://www.xes-standard.org/xesstandarddefinition (besucht am 05. 08. 2014) (siehe S. 176).

[44] Gregor Hohpe und Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321200683 (siehe S. 5, 52, 153, 154).

[45] *IBM WebSphere*. URL: http://www-01.ibm.com/software/de/websphere/ (besucht am 03. 07. 2014) (siehe S. 107).

[46] *iptables*. URL: http://www.netfilter.org/projects/iptables (besucht am 18. 08. 2014) (siehe S. 87).

[47] Kurt Jensen und Lars Michael Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009, S. I–XI, 1–384. ISBN: 978-3-642-00283-0 (siehe S. 154, 159).

[48] Ekkart Kindler. „A Compositional Partial Order Semantics for Petri Net Components“. In: *ICATPN*. 1997, S. 235–252 (siehe S. 30).

[49] Jens Kleine. „Transformation von offenen Workflow-Netzen zu abstrakten WS-BPEL-Prozessen“. Diplomarbeit. Humboldt-Universität zu Berlin, Juli 2007 (siehe S. 108, 110).

[50] Woralak Kongdenfha u. a. „An Aspect-Oriented Framework for Service Adaptation“. In: *Service-Oriented Computing - ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings*. Hrsg. von Asit Dan und Winfried Lamersdorf. Springer, 2006, S. 15–26. DOI: 10.1007/11948148_2. URL: http://dx.doi.org/10.1007/11948148_2 (siehe S. 8).

[51] Woralak Kongdenfha u. a. „Web Service Adaptation: Mismatch Patterns and Semi-Automated Approach to Mismatch Identification and Adapter Development“. In: *Web Services Foundations*. 2014, S. 245–272. DOI: 10.1007/978-1-4614-7518-7_10. URL: http://dx.doi.org/10.1007/978-1-4614-7518-7_10 (siehe S. 8).

[52] Akhil Kumar und Zhe Shan. „Algorithms Based on Pattern Analysis for Verification and Adapter Creation for Business Process Composition“. In: *On the Move to Meaningful Internet Systems: OTM 2008, OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008, Monterrey, Mexico, November 9-14, 2008, Proceedings, Part I*. Hrsg. von Robert Meersman und Zahir Tari. Springer, 2008, S. 120–138. DOI: 10.1007/978-3-540-88871-0_11. URL: http://dx.doi.org/10.1007/978-3-540-88871-0_11 (siehe S. 9).

[53] Hyun Jung La und Soo Dong Kim. „Adapter Patterns for Resolving Mismatches in Service Discovery“. In: *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops - International Workshops, ICSOC/ServiceWave 2009, Stockholm, Sweden, November 23-27, 2009, Revised Selected Papers*. Hrsg. von Asit Dan, Frederic Gittler und Farouk Toumani. 2009, S. 498–508. DOI: 10.1007/978-3-642-16132-2_47. URL: http://dx.doi.org/10.1007/978-3-642-16132-2_47 (siehe S. 8).

[54] Andreas Lehmann. „Dekomposition von Geschäftsprozessmodellen“. Diplomarbeit. Rostock, Germany: Universität Rostock, Dez. 2010 (siehe S. 13, 123).

[55] Xitong Li u. a. „A pattern-based approach to protocol mediation for web services composition“. In: *Information & Software Technology* 52.3 (2010), S. 304–323. DOI: 10.1016/j.infsof.2009.11.002. URL: http://dx.doi.org/10.1016/j.infsof.2009.11.002 (siehe S. 14).

[56] Niels Lohmann. „A Feature-Complete Petri Net Semantics for WS-BPEL 2.0“. In: *WS-FM*. 2007, S. 77–91 (siehe S. 107).

[57] Niels Lohmann. „Correctness of services and their composition“. Diss. Universität Rostock / Technische Universiteit Eindhoven, 2010, S. 1–189. ISBN: 978-90-386-2318-4 (siehe S. 21, 37, 44).

[58] Niels Lohmann. „Why Does My Service Have No Partners?“ In: *WS-FM*. 2008, S. 191–206 (siehe S. 139, 141).

[59] Niels Lohmann, Christian Gierds und Martin Znamirowski. *Gnu BPEL2oWFN*. 9. Juni 2007. URL: http://www.gnu.org/software/bpel2owfn/ (besucht am 03. 07. 2014) (siehe S. 108).

[60] Niels Lohmann, Peter Massuthe und Karsten Wolf. „Behavioral Constraints for Services“. In: *BPM*. 2007, S. 271–287 (siehe S. 95).

[61] Niels Lohmann, Peter Massuthe und Karsten Wolf. „Operating Guidelines for Finite-State Services“. In: *Petri Nets and Other Models of Concurrency - ICATPN 2007, 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, ICATPN 2007, Siedlce, Poland, June 25-29, 2007, Proceedings*. Hrsg. von Jetty Kleijn und Alexandre Yakovlev. Springer, 2007, S. 321–341. DOI: 10.1007/978-3-540-73094-1_20. URL: http://dx.doi.org/10.1007/978-3-540-73094-1_20 (siehe S. 5, 16, 31, 36, 44, 95, 102).

[62] David L. Martin u. a. „Bringing Semantics to Web Services with OWL-S“. In: *World Wide Web*. 2007, S. 243–277 (siehe S. 5, 7).

[63] José Antonio Martín und Ernesto Pimentel. „Automatic Generation of Adaptation Contracts“. In: *Electr. Notes Theor. Comput. Sci.* 229.2 (2009), S. 115–131. DOI: 10.1016/j.entcs.2009.06.032. URL: http://dx.doi.org/10.1016/j.entcs.2009.06.032 (siehe S. 11).

[64] Peter Massuthe. „Operating guidelines for services.“ Diss. Humboldt-Universität zu Berlin / Technische Universiteit Eindhoven, 2009 (siehe S. 37, 44).

[65] Radu Mateescu, Pascal Poizat und Gwen Salaün. „Adaptation of Service Protocols Using Process Algebra and On-the-Fly Reduction Techniques“. In: *IEEE Trans. Software Eng.* 38.4 (2012), S. 755–777. DOI: 10.1109/TSE.2011.62. URL: http://doi.ieeecomputersociety.org/10.1109/TSE.2011.62 (siehe S. 11).

[66] Brahim Medjahed u. a. „Business-to-business interactions: issues and enabling technologies“. In: *VLDB J.* 12.1 (2003), S. 59–85. DOI: 10.1007/s00778-003-0087-z. URL: http://dx.doi.org/10.1007/s00778-003-0087-z (siehe S. 6).

[67] Tarek Melliti, Pascal Poizat und Sonia Ben Mokhtar. „Distributed Behavioural Adaptation for the Automatic Composition of Semantic Services“. In: *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. 2008, S. 146–162. DOI: 10.1007/978-3-540-78743-3_12. URL: http://dx.doi.org/10.1007/978-3-540-78743-3_12 (siehe S. 13).

[68] Stephan Mennicke, Olivia Oanea und Karsten Wolf. „Decomposition into open nets“. In: *16th German Workshop on Algorithms and Tools for Petri Nets, AWPN 2009, Karlsruhe, Germany, September 25, 2009, Proceedings*. Hrsg. von Thomas Freytag und Andreas Eckleder. Bd. 501. CEUR Workshop Proceedings. CEUR-WS.org, Sep. 2009, S. 29–34 (siehe S. 13).

[69] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989. ISBN: 978-0-13-115007-2 (siehe S. 10, 21).

[70] Robin Milner, Joachim Parrow und David Walker. „A Calculus of Mobile Processes, I“. In: *Inf. Comput.* 100.1 (1992), S. 1–40 (siehe S. 10).

[71] Michael K. Molloy. „Performance Analysis Using Stochastic Petri Nets“. In: *IEEE Trans. Computers* 31.9 (1982), S. 913–917. DOI: 10.1109/TC.1982.1676110. URL: http://doi.ieeecomputersociety.org/10.1109/TC.1982.1676110 (siehe S. 187).

[72] Arjan J. Mooij und Marc Voorhoeve. „Proof Techniques for Adapter Generation“. In: *WS-FM*. 2008, S. 207–223 (siehe S. 79).

[73] Arjan J. Mooij und Marc Voorhoeve. „Specification and Generation of Adapters for System Integration“. In: *Situation Awareness with Systems of Systems*. 2013, S. 173–187. DOI: 10.1007/978-1-4614-6230-9_11. URL: http://dx.doi.org/10.1007/978-1-4614-6230-9_11 (siehe S. 12).

[74] Arjan J. Mooij und Marc Voorhoeve. „Trading Off Concurrency to Generate Behavioral Adapters“. In: *ACSD*. 2009, S. 109–118 (siehe S. 73, 80, 82).

[75] Arjan J. Mooij u. a. „Constructing Replaceable Services Using Operating Guidelines and Maximal Controllers“. In: *Web Services and Formal Methods - 7th International Workshop, WS-FM 2010, Hoboken, NJ, USA, September 16-17, 2010. Revised Selected Papers*. Hrsg. von Mario Bravetti und Tevfik Bultan. Springer, 2010, S. 116–130. DOI: 10.1007/978-3-642-19589-1_8. URL: http://dx.doi.org/10.1007/978-3-642-19589-1_8 (siehe S. 186).

[76] Hamid R. Motahari Nezhad, Guang Yuan Xu und Boualem Benatallah. „Protocol-aware matching of web service interfaces for adapter development“. In: *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*. Hrsg. von Michael Rappa u. a. ACM, 2010, S. 731–740. DOI: 10.1145/1772690.1772765. URL: http://doi.acm.org/10.1145/1772690.1772765 (siehe S. 11).

[77] Hamid R. Motahari Nezhad u. a. „Event correlation for process discovery from web service interaction logs“. In: *VLDB J.* 20.3 (2011), S. 417–444. DOI: 10.1007/s00778-010-0203-9. URL: http://dx.doi.org/10.1007/s00778-010-0203-9 (siehe S. 179).

[78] Hamid R. Motahari Nezhad u. a. „Semi-automated adaptation of service interactions“. In: *WWW*. 2007, S. 993–1002 (siehe S. 10, 12, 14, 52).

[79] Richard Müller u. a. „Service Discovery from Observed Behavior while Guaranteeing Deadlock Freedom in Collaborations“. In: *ICSOC*. 2013, S. 358–373 (siehe S. 14, 159, 179).

[80] Tadao Murata. „Petri nets: Properties, analysis and applications“. In: *Proceedings of the IEEE* 77 (4 Apr. 1989), S. 541–580. ISSN: 0018-9219. DOI: 10.1109/5.24143 (siehe S. 77, 80, 136).

[81] Kreshnik Musaraj u. a. „Message Correlation and Web Service Protocol Mining from Inaccurate Logs“. In: *IEEE International Conference on Web Services, ICWS 2010, Miami, Florida, USA, July 5-10, 2010*. IEEE Computer Society, 2010, S. 259–266. DOI: 10.1109/ICWS.2010.104. URL: http://doi.ieeecomputersociety.org/10.1109/ICWS.2010.104 (siehe S. 14).

[82] Hamid R. Motahari Nezhad u. a. „Deriving Protocol Models from Imperfect Service Conversation Logs“. In: *IEEE Trans. Knowl. Data Eng.* 20.12 (2008), S. 1683–1698. DOI: 10.1109/TKDE.2008.87. URL: http://dx.doi.org/10.1109/TKDE.2008.87 (siehe S. 14).

[83] OASIS. *Web Services Business Process Execution Language Version 2.0*. Hrsg. von Alexandre Alves u. a. 11. Apr. 2007. URL: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html (besucht am 03. 07. 2014) (siehe S. 107).

[84] *Oracle BPEL Process Manager*. URL: http://www.oracle.com/us/products/middleware/soa/bpel-process-manager/overview/index.html (besucht am 03. 07. 2014) (siehe S. 107).

[85] Michael P. Papazoglou. *Web Services - Principles and Technology*. Prentice Hall, 2008. ISBN: 978-0-321-15555-9. URL: http://vig.pearsoned.com/store/product/1,1207,store-12521_isbn-0321155556,00.html (siehe S. 1, 6).

[86] Chris Peltz. „Web Services Orchestration and Choreography“. In: *IEEE Computer* 36.10 (2003), S. 46–52. DOI: 10.1109/MC.2003.1236471. URL: http://doi.ieeecomputersociety.org/10.1109/MC.2003.1236471 (siehe S. 11).

[87] Amir Pnueli und Roni Rosner. „On the Synthesis of an Asynchronous Reactive Module“. In: *Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11-15, 1989, Proceedings*. Hrsg. von Giorgio Ausiello, Mariangiola Dezani-Ciancaglini und Simona Ronchi Della Rocca. Springer, 1989, S. 652–671. DOI: 10.1007/BFb0035790. URL: http://dx.doi.org/10.1007/BFb0035790 (siehe S. 5).

[88] Shankar Ponnekanti und Armando Fox. „Interoperability Among Independently Evolving Web Services“. In: *Middleware 2004, ACM/IFIP/USENIX International Middleware Conference, Toronto, Canada, October 18-20, 2004, Proceedings*. Hrsg. von Hans-Arno Jacobsen. Springer, 2004, S. 331–351. DOI: 10.1007/978-3-540-30229-2_18. URL: http://dx.doi.org/10.1007/978-3-540-30229-2_18 (siehe S. 7).

[89] Erhard Rahm und Philip A. Bernstein. „A survey of approaches to automatic schema matching“. In: *VLDB J.* 10.4 (2001), S. 334–350. DOI: 10.1007/s007780100057. URL: http://dx.doi.org/10.1007/s007780100057 (siehe S. 7).

[90] Jinghai Rao und Xiaomeng Su. „A Survey of Automated Web Service Composition Methods“. In: *Semantic Web Services and Web Process Composition, First International Workshop, SWSWPC 2004, San Diego, CA, USA, July 6, 2004, Revised Selected Papers*. Hrsg. von Jorge Cardoso und Amit P. Sheth. Springer, 2004, S. 43–54. DOI: 10.1007/978-3-540-30581-1_5. URL: http://dx.doi.org/10.1007/978-3-540-30581-1_5 (siehe S. 15).

[91] Wolfgang Reisig. *Petrinetze, Eine Einführung*. Springer, 1982. ISBN: 3-540-11478-5 (siehe S. 23).

[92] Wolfgang Reisig. *Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies*. Springer, 2013, S. I–XXVI, 1–230. ISBN: 978-3-642-33277-7 (siehe S. 77).

[93] Daniel Ritter. „Experiences with Business Process Model and Notation for Modeling Integration Patterns“. In: *ECMFA*. 2014, S. 254–266 (siehe S. 5).

[94] Anne Rozinat und Wil M. P. van der Aalst. „Conformance Testing: Measuring the Fit and Appropriateness of Event Logs and Process Models“. In: *Business Process Management Workshops*. 2005, S. 163–176 (siehe S. 158).

[95] Anne Rozinat u. a. „Discovering colored Petri nets from event logs“. In: *International Journal on Software Tools for Technology Transfer* 10.1 (2008), S. 57–74 (siehe S. 14).

[96] Gwen Salaün. „Generation of Service Wrapper Protocols from Choreography Specifications“. In: *Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa, 10-14 November 2008*. Hrsg. von Antonio Cerone und Stefan Gruner. IEEE Computer Society, 2008, S. 313–322. DOI: 10.1109/SEFM.2008.42. URL: http://dx.doi.org/10.1109/SEFM.2008.42 (siehe S. 13).

[97] R. Seguel, R. Eshuis und P. Grefen. *An Overview on Protocol Adaptors for Service Component Integration*. Technical Report. TU/e, Beta, 2008 (siehe S. 6).

[98] Ricardo Seguel, Rik Eshuis und Paul W. P. J. Grefen. „Generating Minimal Protocol Adaptors for Loosely Coupled Services“. In: *IEEE International Conference on Web Services, ICWS 2010, Miami, Florida, USA, July 5-10, 2010*. IEEE Computer Society, 2010, S. 417–424. DOI: 10.1109/ICWS.2010.14. URL: http://doi.ieeecomputersociety.org/10.1109/ICWS.2010.14 (siehe S. 9).

[99] Zhe Shan und Akhil Kumar. „Optimal Adapter Creation for Process Composition in Synchronous vs. Asynchronous Communication“. In: *ACM Trans. Management Inf. Syst.* 3.2 (2012), S. 8. DOI: 10.1145/2229156.2229160. URL: http://doi.acm.org/10.1145/2229156.2229160 (siehe S. 9).

[100] Christian Stahl und Karsten Wolf. „Covering Places and Transitions in Open Nets“. In: *BPM*. 2008, S. 116–131 (siehe S. 95, 98).

[101] Christian Stahl und Karsten Wolf. „Deciding service composition and substitutability using extended operating guidelines“. In: *Data Knowl. Eng.* 68.9 (2009), S. 819–833. DOI: 10.1016/j.datak.2009.02.012. URL: http://dx.doi.org/10.1016/j.datak.2009.02.012 (siehe S. 186).

[102] Peter H. Starke. *Analyse von Petri-Netz-Modellen*. Leitfäden und Monographien der Informatik. Teubner, 1990 (siehe S. 28).

[103] Jan Sürmeli. „Cost-minimal Adapters for Services“. In: *ZEUS*. 2012, S. 9–16 (siehe S. 104).

[104] Jan Sürmeli und Marvin Triebel. „Synthesizing Cost-Minimal Partners for Services“. In: *ICSOC*. 2013, S. 584–591 (siehe S. 104).

[105] Yehia Taher u. a. „Diagnosing Incompatibilities in Web Service Interactions for Automatic Generation of Adapters“. In: *The IEEE 23rd International Conference on Advanced Information Networking and Applications, AINA 2009, Bradford, United Kingdom, May 26-29, 2009*. Hrsg. von Irfan Awan u. a. IEEE Computer Society, 2009, S. 652–659. DOI: 10.1109/AINA.2009.118. URL: http://dx.doi.org/10.1109/AINA.2009.118 (siehe S. 14).

[106] Roman Vaculín und Katia P. Sycara. „Towards automatic mediation of OWL-S process models“. In: *2007 IEEE International Conference on Web Services (ICWS 2007), July 9-13, 2007, Salt Lake City, Utah, USA*. IEEE Computer Society, 2007, S. 1032–1039. DOI: 10.1109/ICWS.2007.177. URL: http://doi.ieeecomputersociety.org/10.1109/ICWS.2007.177 (siehe S. 14).

[107] W3C. *Extensible Markup Language (XML) 1.0*. Hrsg. von Tim Bray u. a. 26. Nov. 2008. URL: http://www.w3.org/TR/xml/ (besucht am 29. 09. 2014) (siehe S. 7).

[108] W3C. *Soap*. Hrsg. von Martin Gudgin u. a. 27. Apr. 2007. URL: http://www.w3.org/TR/soap12 (besucht am 08. 07. 2014) (siehe S. 112).

[109] W3C. *Web Service Choreography Interface (WSCI) 1.0*. Hrsg. von Assaf Arkin u. a. 8. Aug. 2002. URL: http://www.w3.org/TR/wsci/ (siehe S. 10).

[110] Kenneth Wang u. a. „The Service Adaptation Machine“. In: *ECOWS 2008, Sixth European Conference on Web Services, 12-14 November 2008, Dublin, Ireland*. Hrsg. von Claus Pahl, Siobhán Clarke und Rik Eshuis. IEEE Computer Society, 2008, S. 145–154. DOI: 10.1109/ECOWS.2008.23. URL: http://dx.doi.org/10.1109/ECOWS.2008.23 (siehe S. 11).

[111] Fabian Weber. „Implementierung eines synthetisierten Webservice-Adapters“. Studienarbeit. Humboldt-Universität zu Berlin, 2014 (siehe S. 111).

[112] Daniela Weinberg. „Deciding Service Substitution - Termination guaranteed“. Diss. Universität Rostock, 2012 (siehe S. 36, 37, 44, 95).

[113] Michael Westergaard und Lars Michael Kristensen. *Access/CPN*. URL: http://cpntools.org/accesscpn (besucht am 05. 08. 2014) (siehe S. 176).

[114] Karsten Wolf. „Does My Service Have Partners?“ In: *T. Petri Nets and Other Models of Concurrency* 2 (2009), S. 152–171 (siehe S. 31).

[115] Karsten Wolf u. a. „Guaranteeing Weak Termination in Service Discovery“. In: *Fundam. Inform.* 108.1-2 (2011), S. 151–180 (siehe S. 36, 44).

[116] Xie Xiong und Zhang Weishi. „The Current State of Software Component Adaptation“. In: *2005 International Conference on Semantics, Knowledge and Grid (SKG 2005), 27-29 November 2005, Beijing, China*. IEEE Computer Society, 2005, S. 103. DOI: 10.1109/SKG.2005.123. URL: http://doi.ieeecomputersociety.org/10.1109/SKG.2005.123 (siehe S. 6).

[117] Daniel M. Yellin und Robert E. Strom. „Protocol Specifications and Component Adaptors“. In: *ACM Trans. Program. Lang. Syst.* 19.2 (1997), S. 292–333. DOI: 10.1145/244795.244801. URL: http://doi.acm.org/10.1145/244795.244801 (siehe S. 3, 6).

[118] George Zheng und Athman Bouguettaya. „Service Mining on the Web“. In: *IEEE T. Services Computing* 2.1 (2009), S. 65–78. DOI: 10.1109/TSC.2009.2. URL: http://doi.ieeecomputersociety.org/10.1109/TSC.2009.2 (siehe S. 15).

[119] ZhangBing Zhou u. a. „Developing Process Mediator for Supporting Mediated Web Service Interactions“. In: *ECOWS 2008, Sixth European Conference on Web Services, 12-14 November 2008, Dublin, Ireland*. Hrsg. von Claus Pahl, Siobhán Clarke und Rik Eshuis. IEEE Computer Society, 2008, S. 155–164. DOI: 10.1109/ECOWS.2008.10. URL: http://dx.doi.org/10.1109/ECOWS.2008.10 (siehe S. 11).

PUBLIKATIONSLISTE

Christian Gierds. „Finding Cost-Efficient Adapters“. In: *AWPN*. (Workshop, mit Vortrag). 2008, S. 37–42.

Christian Gierds und Niels Lohmann. „A Graphical User Interface for Service Adaptation“. In: *Proceedings of the 17th German Workshop on Algorithms and Tools for Petri Nets, AWPN 2010, Cottbus, Germany, October 07-08, 2010*. Hrsg. von Martin Schwarick und Monika Heiner. Bd. 643. CEUR Workshop Proceedings. (Workshop, mit Vortrag). CEUR-WS.org, Okt. 2010, S. 136–141.

Christian Gierds und Jan Sürmeli, Hrsg. *2nd Central-European Workshop on Services and their Composition, Services und ihre Komposition, ZEUS 2010, Berlin, Germany, February 25-26, 2010. Proceedings*. Bd. 563. CEUR Workshop Proceedings. (Workshop Proceedings). CEUR-WS.org, 2010.

Christian Gierds und Jan Sürmeli. „Estimating costs of a service“. In: *ZEUS*. (Workshop). 2010, S. 121–128.

Christian Gierds. „Toward Deciding the Existence of Adaptable Services“. In: *ZEUS*. (Workshop, mit Vortrag). 2012, S. 1–8.

Christian Gierds, Arjan J. Mooij und Karsten Wolf. „Reducing Adapter Synthesis to Controller Synthesis“. In: *IEEE T. Services Computing* 5.1 (2012). (Journal, peer reviewed), S. 72–85.

Dirk Fahland und Christian Gierds. „Analyzing and Completing Middleware Designs for Enterprise Integration Using Coloured Petri Nets“. In: *CAiSE*. (Konferenz, peer reviewed, mit Vortrag). 2013, S. 400–416.

Christian Gierds und Dirk Fahland. „Discovering Pattern-Based Mediator Services from Communication Logs“. In: *ICSOC Workshops*. (Workshop, peer reviewed, mit Vortrag). 2013, S. 123–134.